Compiler M(ang)o

Project Report

YurongYou ACM 2014

Features & Basic Info

Features & Basic Info

- well designed structure

Features & Basic Info

- well designed structure
- effective optimizations

Features & Basic Info

- well designed structure
- effective optimizations

(Graph coloring based reg alloc)

Features & Basic Info

- well designed structure
- effective optimizations

(Graph coloring based reg alloc)

- efficient target code

Features & Basic Info

- well designed structure
- effective optimizations

(Graph coloring based reg alloc)

- efficient target code
- class functions supported

Features & Basic Info

- well designed structure
- effective optimizations

(Graph coloring based reg alloc)

- efficient target code
- class functions supported
- friendly error messages

Features & Basic Info

- well designed structure
- effective optimizations

(Graph coloring based reg alloc)

- efficient target code
- class functions supported
- friendly error messages
- 5292 lines total Java code

Features & Basic Info

- well designed structure
- effective optimizations

(Graph coloring based reg alloc)

- efficient target code
- class functions supported
- friendly error messages
- 5292 lines total Java code

  (not including ANTLR generated code)

Features & Basic Info

- well designed structure
- effective optimizations

(Graph coloring based reg alloc)

- efficient target code
- class functions supported
- friendly error messages
- 5292 lines total Java code

  (not including ANTLR generated code)
- 660 lines handwritten MIPS BuiltIn Library

Features & Basic Info

- well designed structure
- effective optimizations

(Graph coloring based reg alloc)

- efficient target code
- class functions supported
- friendly error messages
- 5292 lines total Java code

  (not including ANTLR generated code)
- 660 lines handwritten MIPS BuiltIn Library
- Nicely Written JavaDoc Documents

Overview

OverView

.mx
source code

OverView

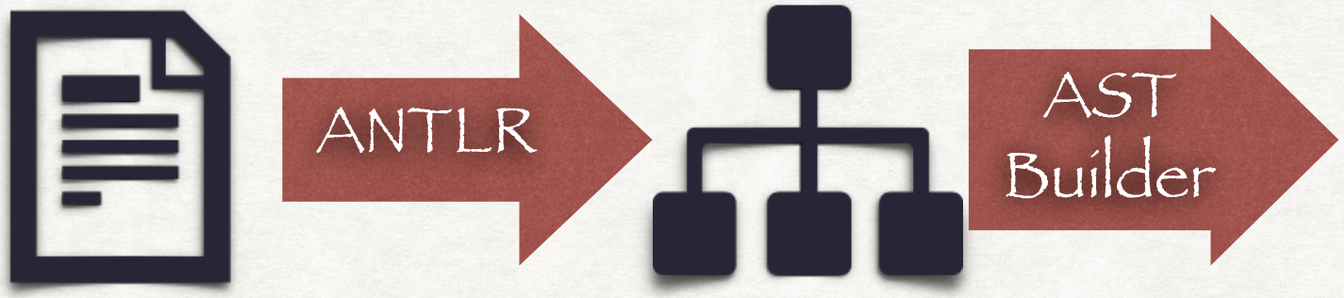
.mx
source code

OverView

.mx
source code

CST

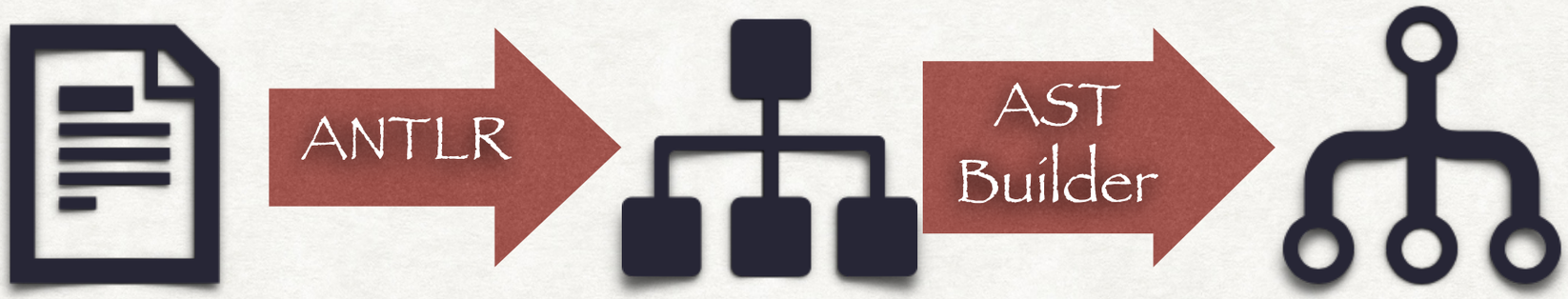
OverView



.mx
source code

CST

OverView



.mx
source code

CST

AST

OverView

.mx
source code

CST

AST

OverView

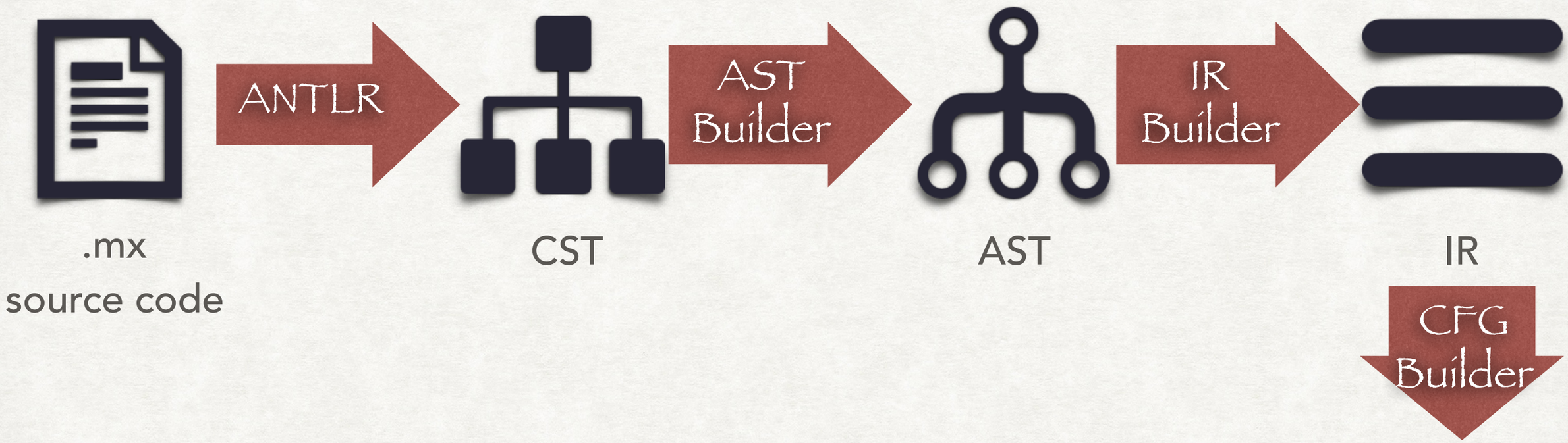
.mx
source code

CST

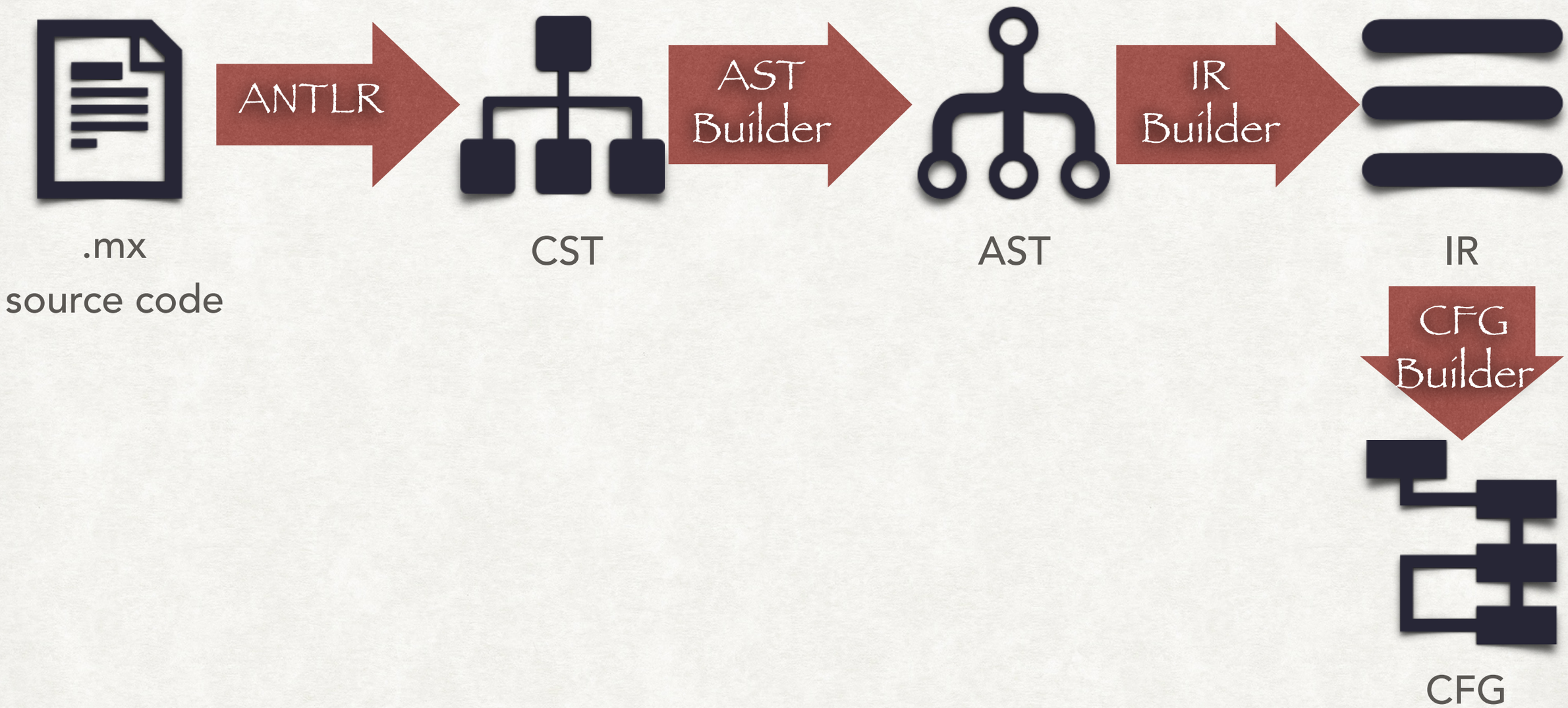
AST

IR

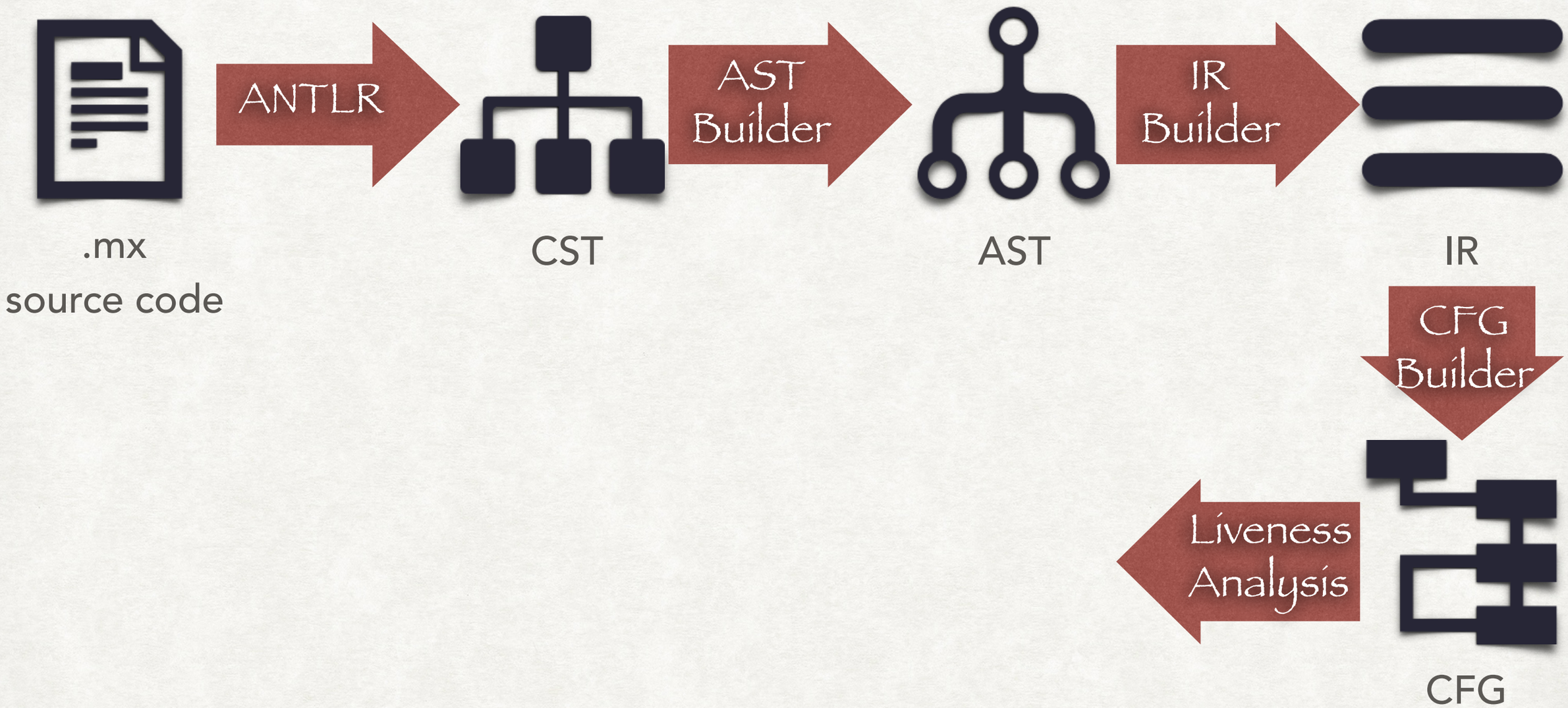
OverView


ANTLR
AST Builder
IR Builder
.mx source code
CST
AST
IR
CFG Builder

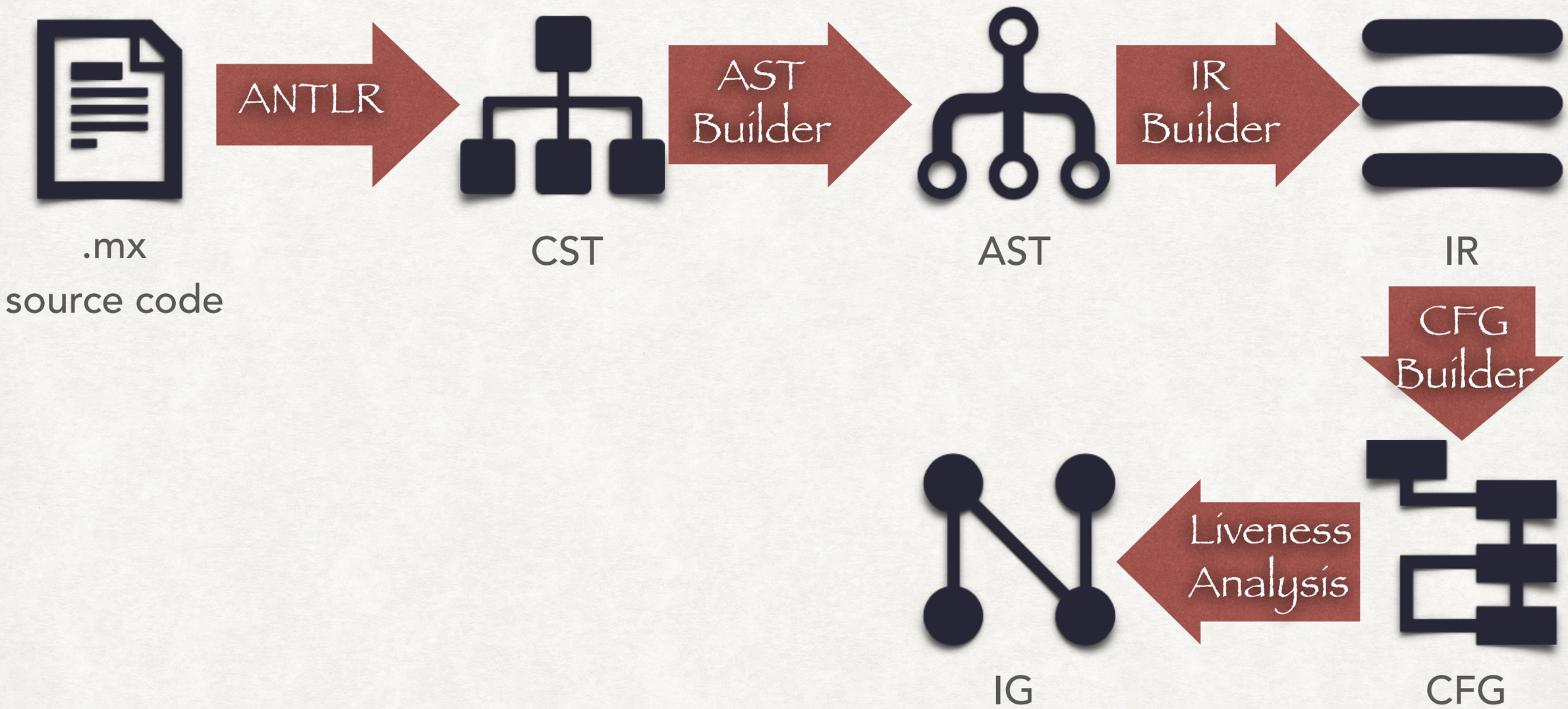
OverView


ANTLR
AST Builder
IR Builder
.mx source code
CST
AST
IR
CFG Builder
CFG

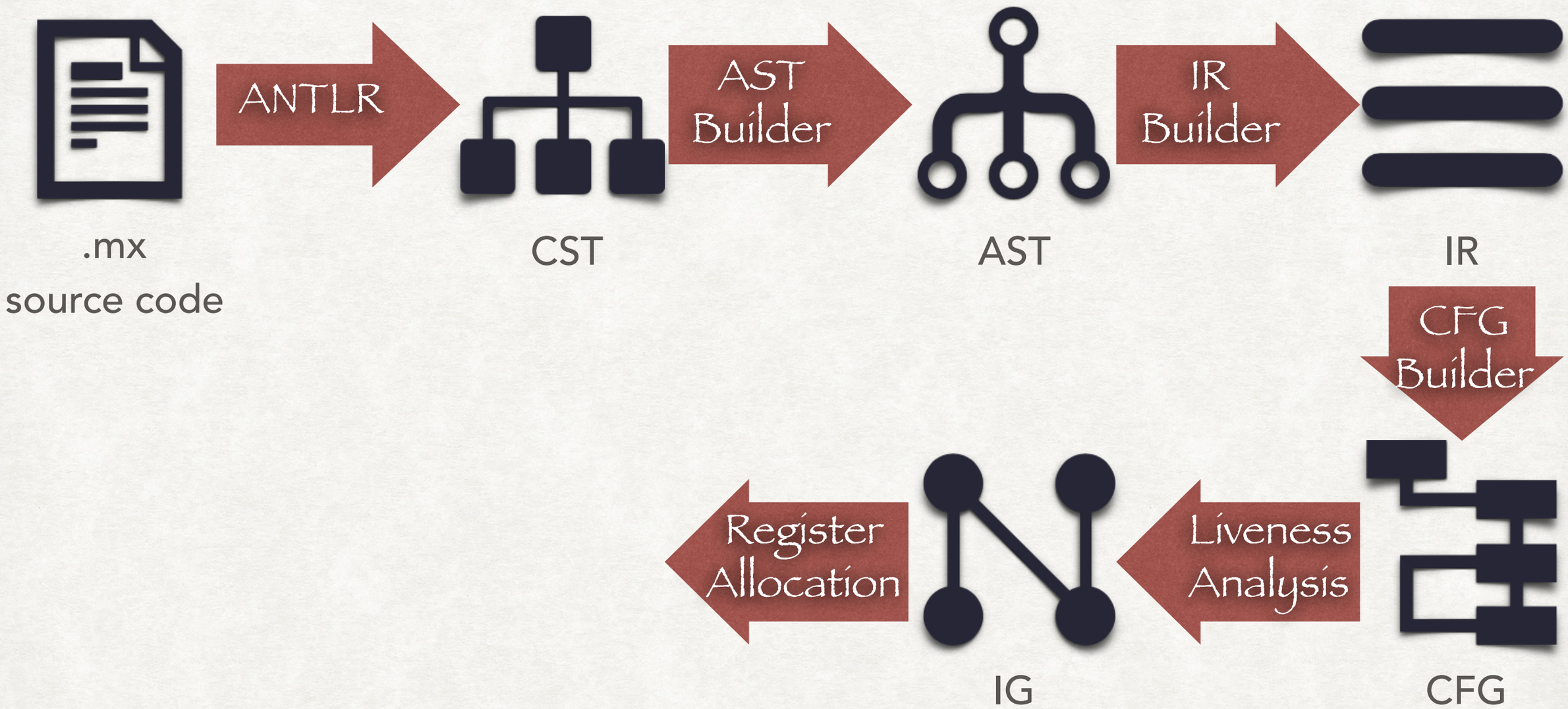
OverView


ANTLR
AST Builder
IR Builder
.mx source code
CST
AST
IR
CFG Builder
Liveness Analysis
CFG

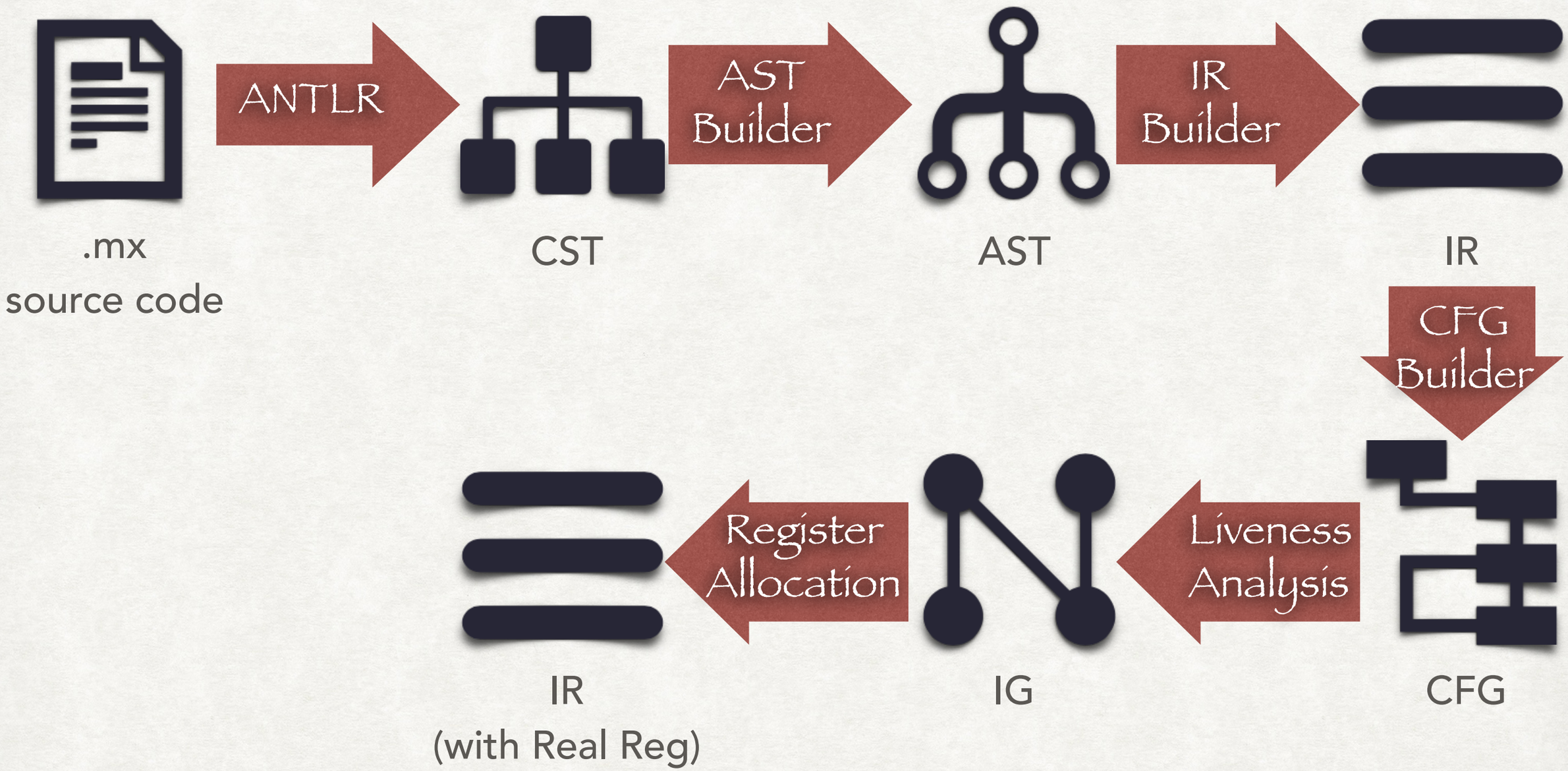
OverView


ANTLR
AST Builder
IR Builder
.mx source code
CST
AST
IR
CFG Builder
Liveness Analysis
IG
CFG

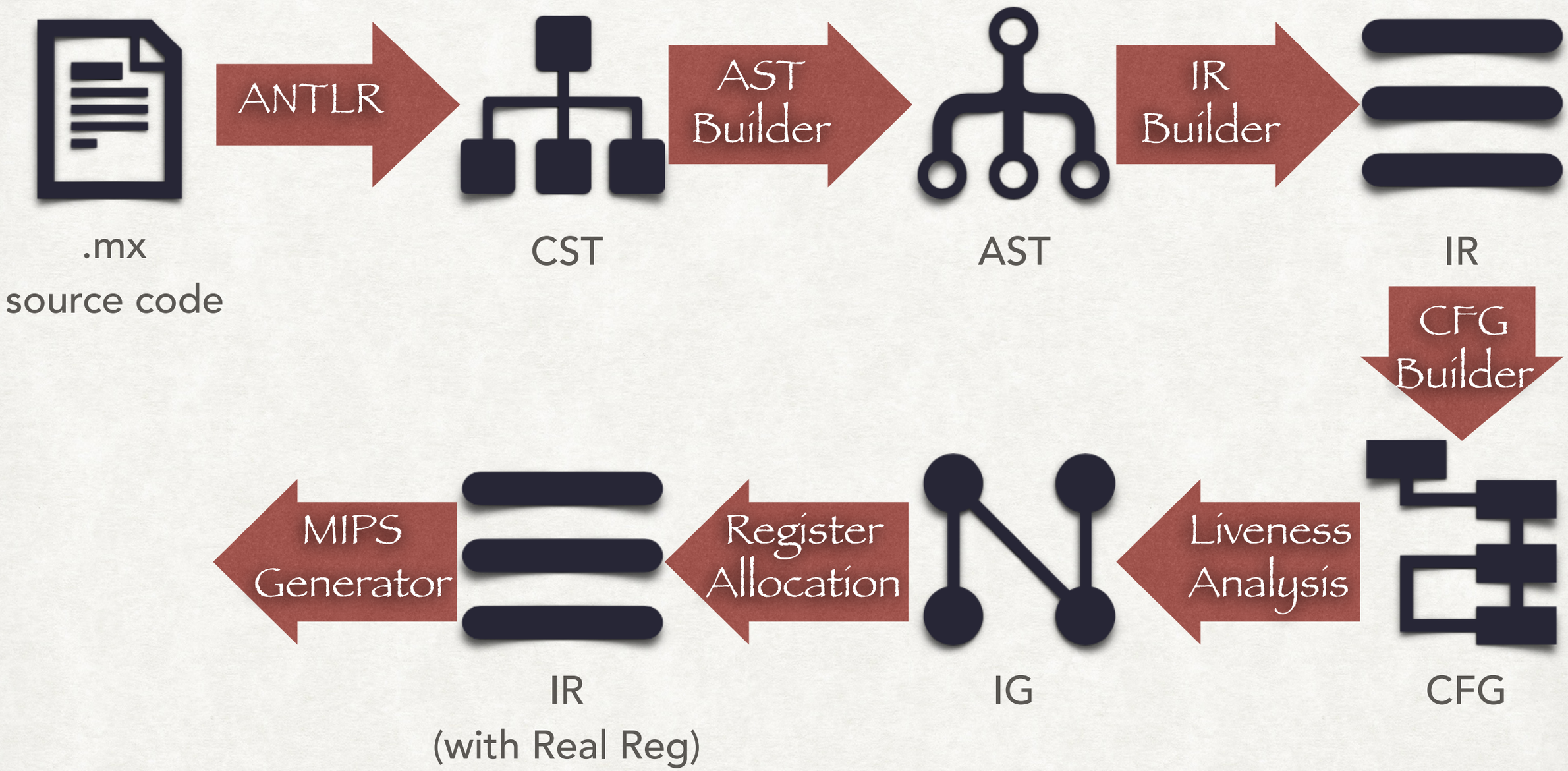
OverView


ANTLR
AST Builder
IR Builder
.mx source code
CST
AST
IR
CFG Builder
Register Allocation
Liveness Analysis
IG
CFG

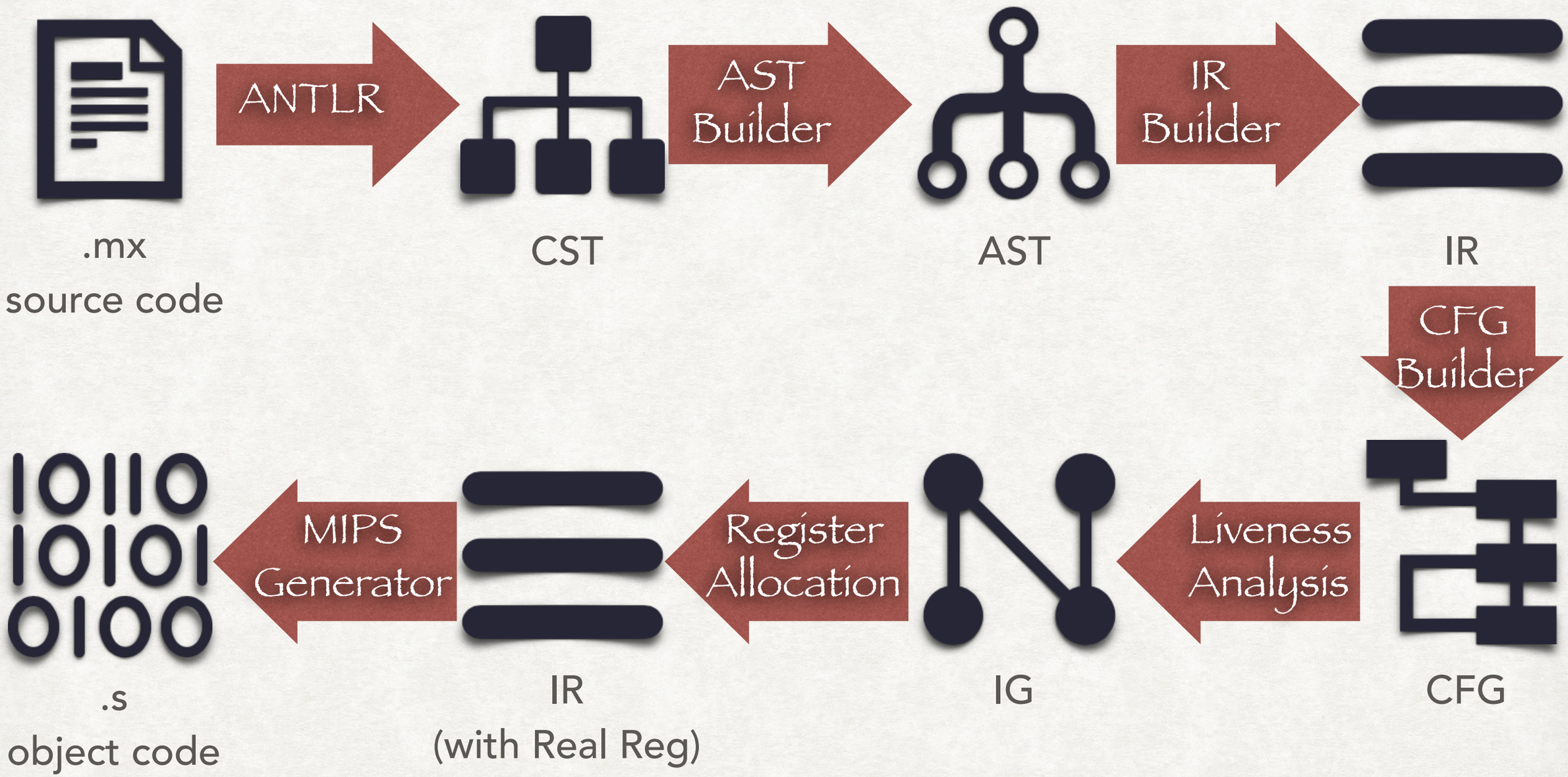
OverView


ANTLR
AST Builder
IR Builder
.mx source code
CST
AST
IR
CFG Builder
Register Allocation
Liveness Analysis
IR (with Real Reg)
IG
CFG

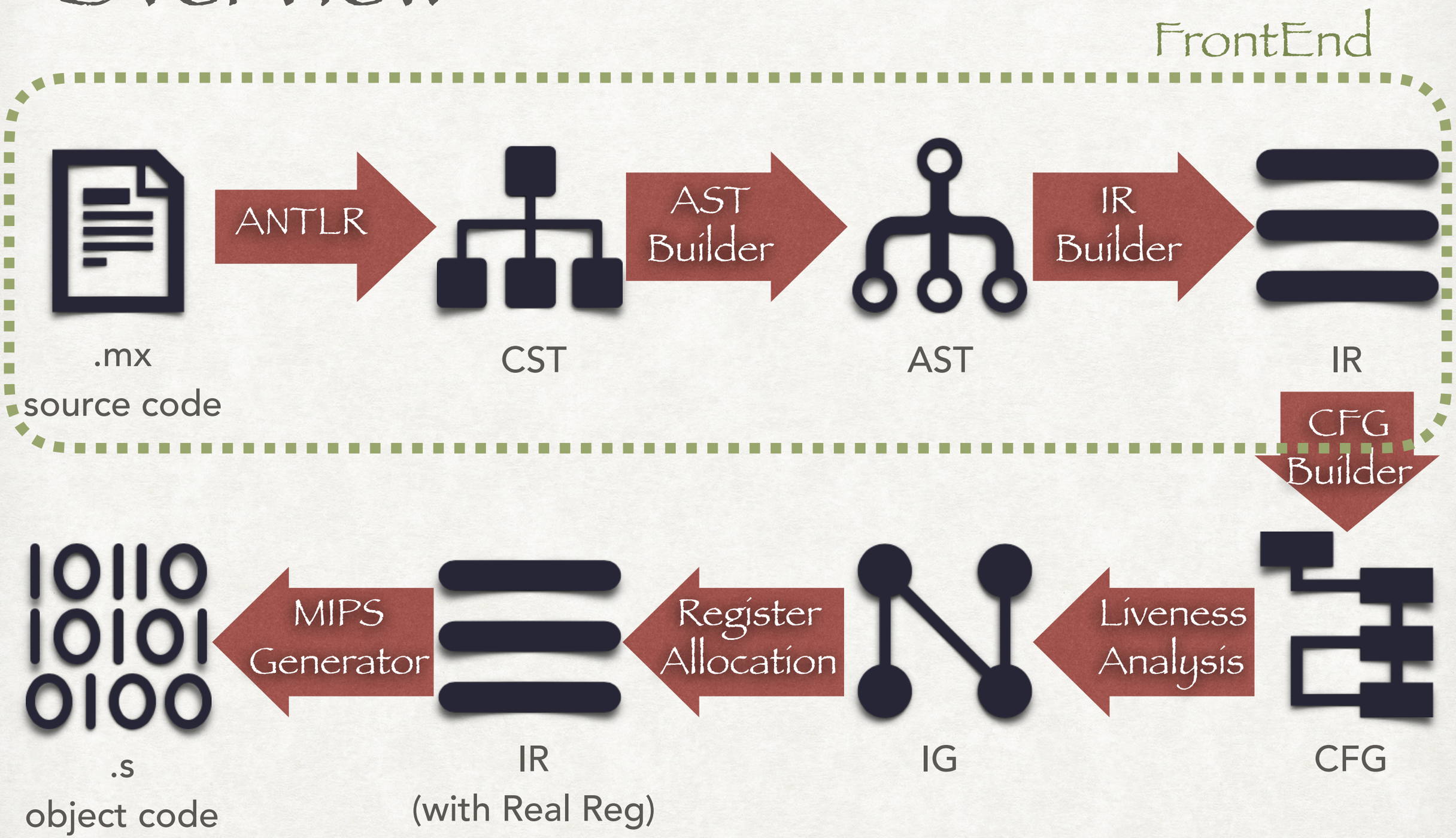
OverView


.mx
source code
ANTLR
CST
AST
Builder
AST
IR
Builder
IR
CFG
Builder
CFG
Liveness
Analysis
IG
Register
Allocation
IR
(with Real Reg)
MIPS
Generator

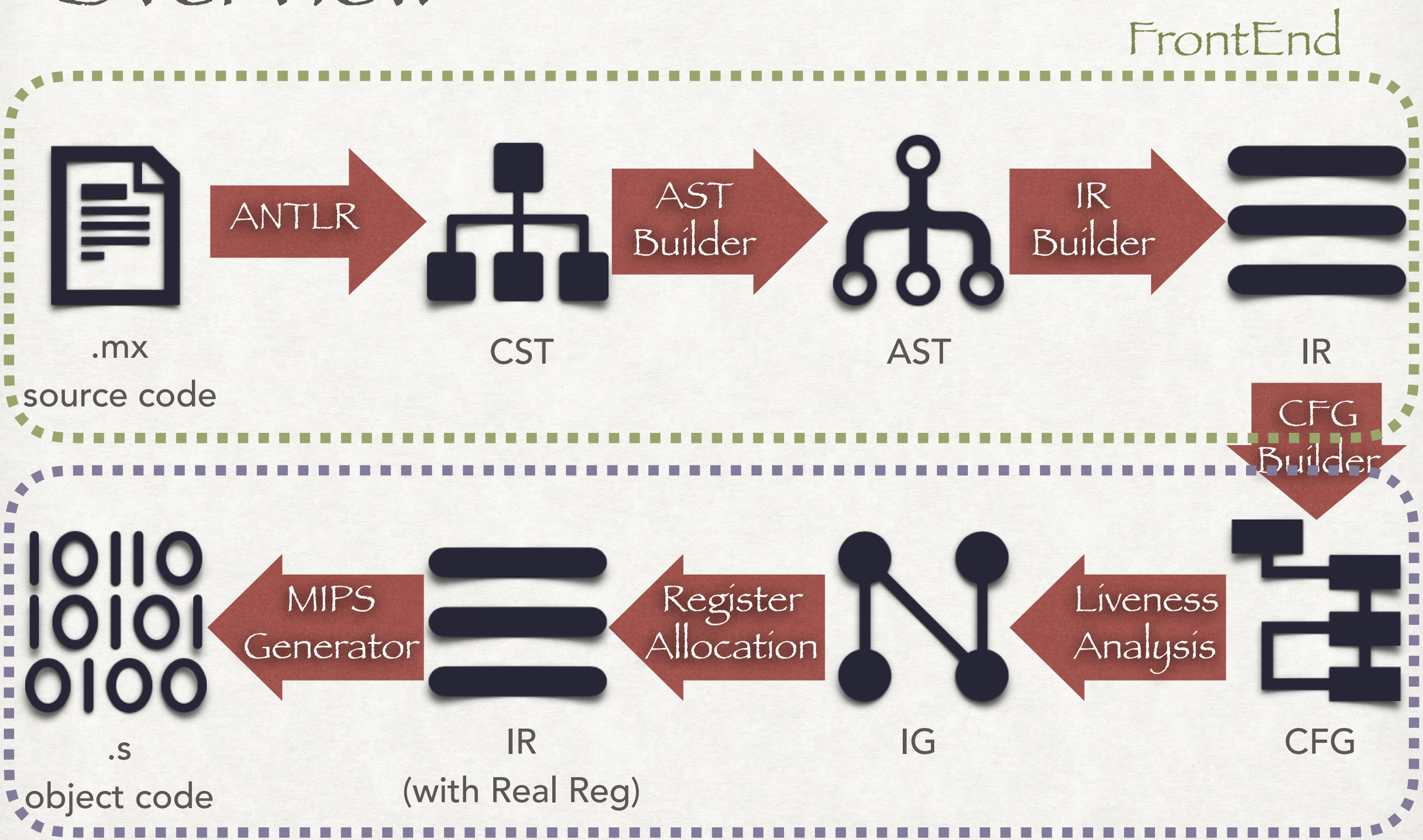
OverView


.mx
source code
ANTLR
CST
AST
Builder
AST
IR
Builder
IR
CFG
Builder
CFG
Liveness
Analysis
IG
Register
Allocation
IR
(with Real Reg)
MIPS
Generator
.s
object code

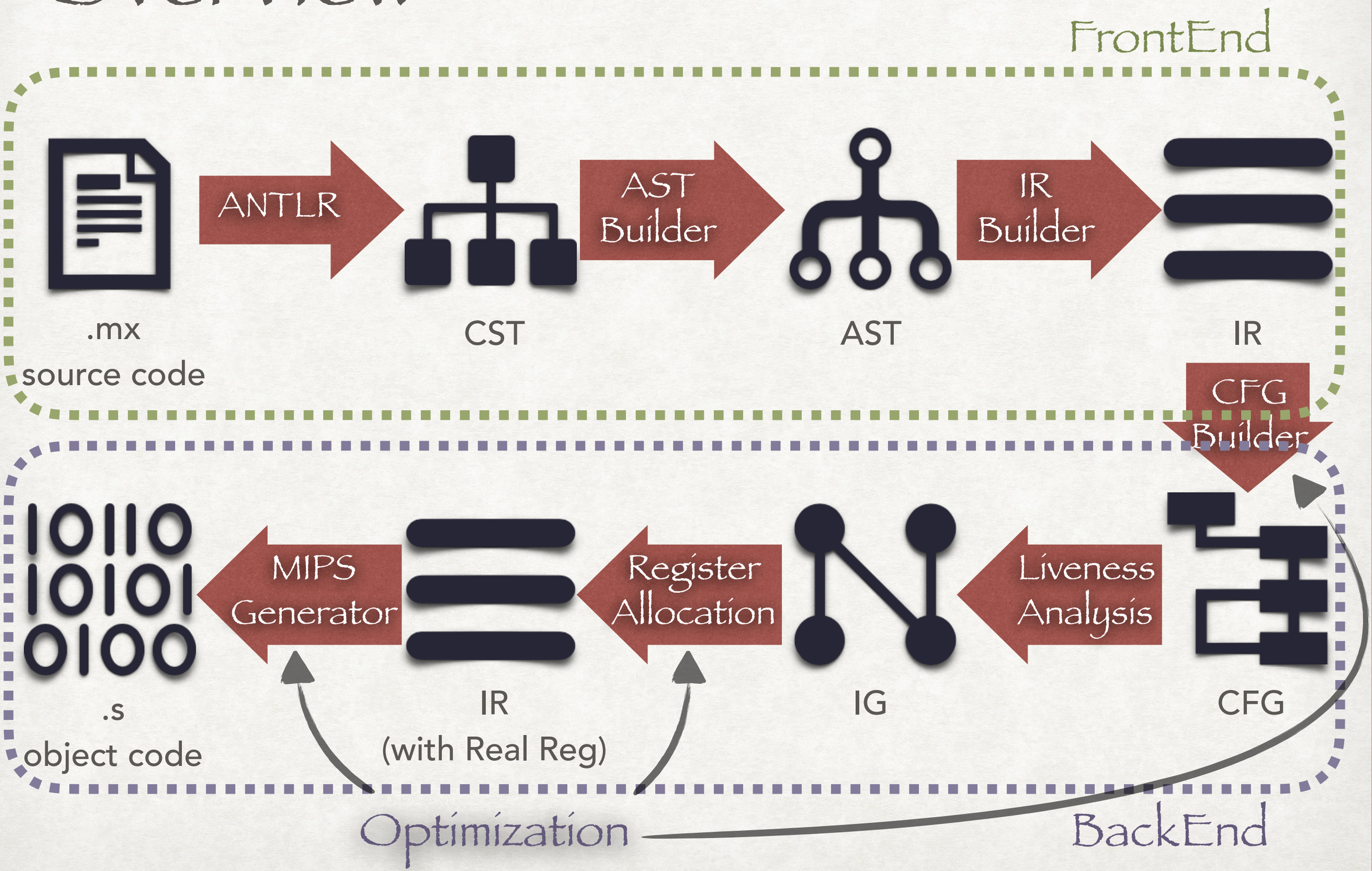
OverView


FrontEnd
ANTLR
AST Builder
IR Builder
.mx
source code
CST
AST
IR
CFG Builder
MIPS Generator
Register Allocation
Liveness Analysis
.s
object code
IR
(with Real Reg)
IG
CFG

OverView


FrontEnd
ANTLR
AST Builder
IR Builder
.mx
source code
CST
AST
IR
CFG Builder
MIPS Generator
Register Allocation
Liveness Analysis
.s
object code
IR
(with Real Reg)
IG
CFG
BackEnd

OverView


FrontEnd
ANTLR
AST Builder
IR Builder
.mx
source code
CST
AST
IR
CFG Builder
MIPS Generator
Register Allocation
Liveness Analysis
.s
object code
IR
(with Real Reg)
IG
CFG
Optimization
BackEnd

Structure: Lexer & Parser

Structure: Lexer & Parser

- ANTLR generated Lexer & Parser

Structure: Lexer & Parser

- ANTLR generated Lexer & Parser
- BNF with the support to direct Left-Recursion and explicit precedence

```
expr:
    '(' expr ')'                                  # Bracket
    | constant                                    # ConstantLeaf
    | ID                                          # IDLeaf
    | expr op = (PPLUS|MMINUS)                    # SelfOpPost
    | ID '(' exprList? ')'                        # Call
    | expr '[' expr ']'                           # Index
    | expr '.' ID                                 # FieldAccess
    | expr '.' ID '(' exprList? ')'               # ClassFuncAccess
    | op = (PPLUS|MMINUS) expr                    # SelfOpPre
    | op = (PLUS | MINUS) expr                    # Sign
    | op = LOG_NOT expr                           # LogNot
    | op = BIT_NOT expr                           # BitNot
    | creationExpr                                # Creation
    | expr op = (MULT | DIV | MOD) expr           # binary
    | expr op = (PLUS | MINUS) expr               # binary
    | expr op = (SHIFT_L | SHIFT_R) expr          # binary
    | expr op = (LESS|LARGE|LEQ|GEQ) expr         # LogBinary
    | expr op = (EQ | NEQ) expr                   # LogBinary
    | expr op = BIT_AND expr                      # binary
    | expr op = BIT_XOR expr                      # binary
    | expr op = BIT_OR expr                       # binary
    | expr op = LOG_AND expr                      # LogBinary
    | expr op = LOG_OR expr                       # LogBinary
    | <assoc=right>expr '=' expr                  # Assign
;
```

Structure: Lexer & Parser

- ANTLR generated Lexer & Parser
- BNF with the support to direct Left-Recursion and explicit precedence
- Visitor & Listener Pattern

```
expr:
    '(' expr ')'                                  # Bracket
    | constant                                    # ConstantLeaf
    | ID                                          # IDLeaf
    | expr op = (PPLUS|MMINUS)                    # SelfOpPost
    | ID '(' exprList? ')'                        # Call
    | expr '[' expr ']'                           # Index
    | expr '.' ID                                 # FieldAccess
    | expr '.' ID '(' exprList? ')'               # ClassFuncAccess
    | op = (PPLUS|MMINUS) expr                    # SelfOpPre
    | op = (PLUS | MINUS) expr                    # Sign
    | op = LOG_NOT expr                           # LogNot
    | op = BIT_NOT expr                           # BitNot
    | creationExpr                                # Creation
    | expr op = (MULT | DIV | MOD) expr           # binary
    | expr op = (PLUS | MINUS) expr               # binary
    | expr op = (SHIFT_L | SHIFT_R) expr          # binary
    | expr op = (LESS|LARGE|LEQ|GEQ) expr         # LogBinary
    | expr op = (EQ | NEQ) expr                   # LogBinary
    | expr op = BIT_AND expr                      # binary
    | expr op = BIT_XOR expr                      # binary
    | expr op = BIT_OR expr                       # binary
    | expr op = LOG_AND expr                      # LogBinary
    | expr op = LOG_OR expr                       # LogBinary
    | <assoc=right>expr '=' expr                  # Assign
;
```

Structure: Lexer & Parser

- ANTLR generated Lexer & Parser
- BNF with the support to direct Left-Recursion and explicit precedence
- Visitor & Listener Pattern
- ANTLR is really an efficient developing tool!

```
expr:
    '(' expr ')'                                  # Bracket
    | constant                                    # ConstantLeaf
    | ID                                          # IDLeaf
    | expr op = (PPLUS|MMINUS)                    # SelfOpPost
    | ID '(' exprList? ')'                        # Call
    | expr '[' expr ']'                           # Index
    | expr '.' ID                                 # FieldAccess
    | expr '.' ID '(' exprList? ')'               # ClassFuncAccess
    | op = (PPLUS|MMINUS) expr                    # SelfOpPre
    | op = (PLUS | MINUS) expr                    # Sign
    | op = LOG_NOT expr                           # LogNot
    | op = BIT_NOT expr                           # BitNot
    | creationExpr                                # Creation
    | expr op = (MULT | DIV | MOD) expr           # binary
    | expr op = (PLUS | MINUS) expr               # binary
    | expr op = (SHIFT_L | SHIFT_R) expr          # binary
    | expr op = (LESS|LARGE|LEQ|GEQ) expr         # LogBinary
    | expr op = (EQ | NEQ) expr                   # LogBinary
    | expr op = BIT_AND expr                      # binary
    | expr op = BIT_XOR expr                      # binary
    | expr op = BIT_OR expr                       # binary
    | expr op = LOG_AND expr                      # LogBinary
    | expr op = LOG_OR expr                       # LogBinary
    | <assoc=right>expr '=' expr                  # Assign
;
```

Structure: AST

Structure: AST

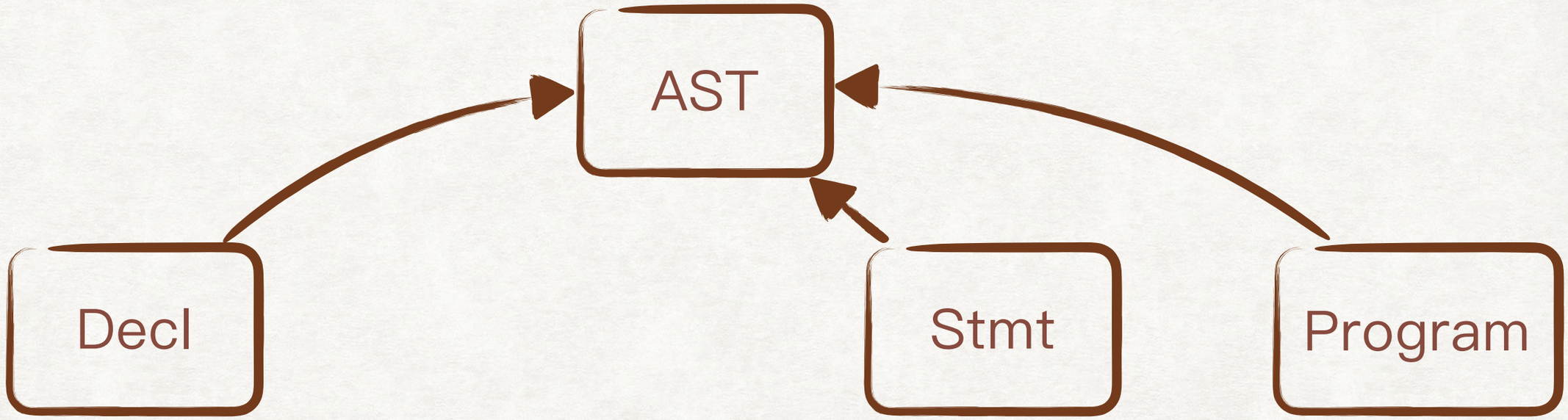
A → B: A extends B

Structure: AST

A → B: A extends B

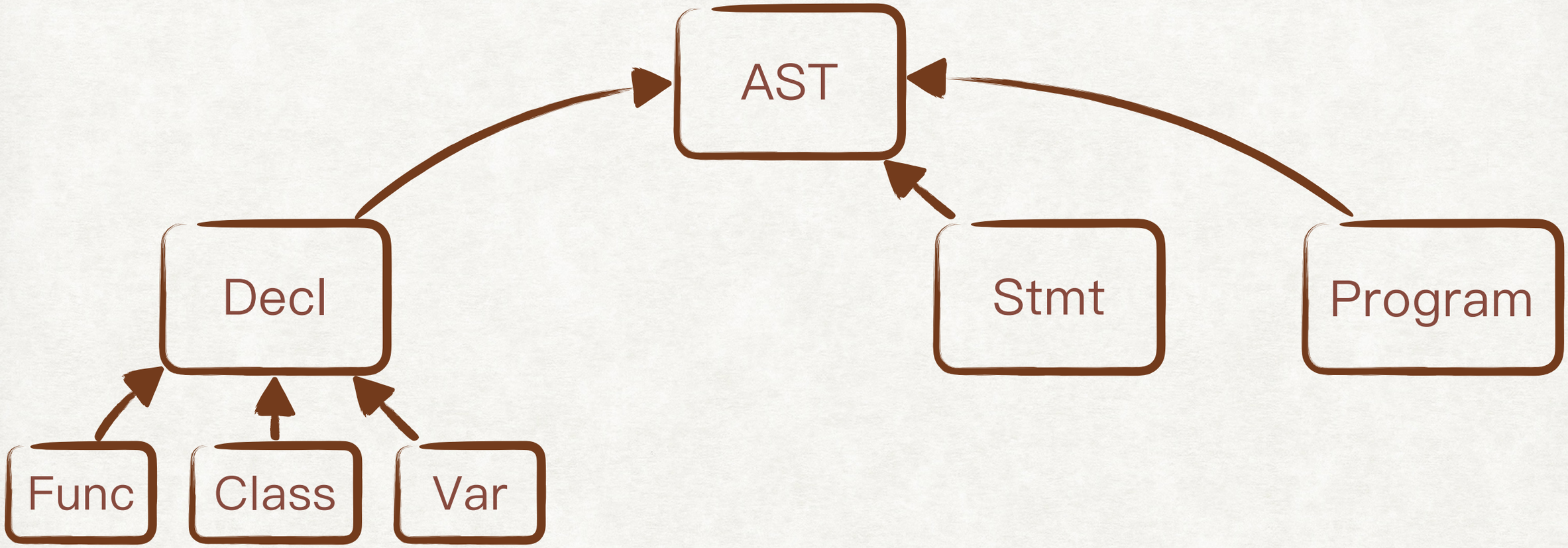
Structure: AST

A → B: A extends B



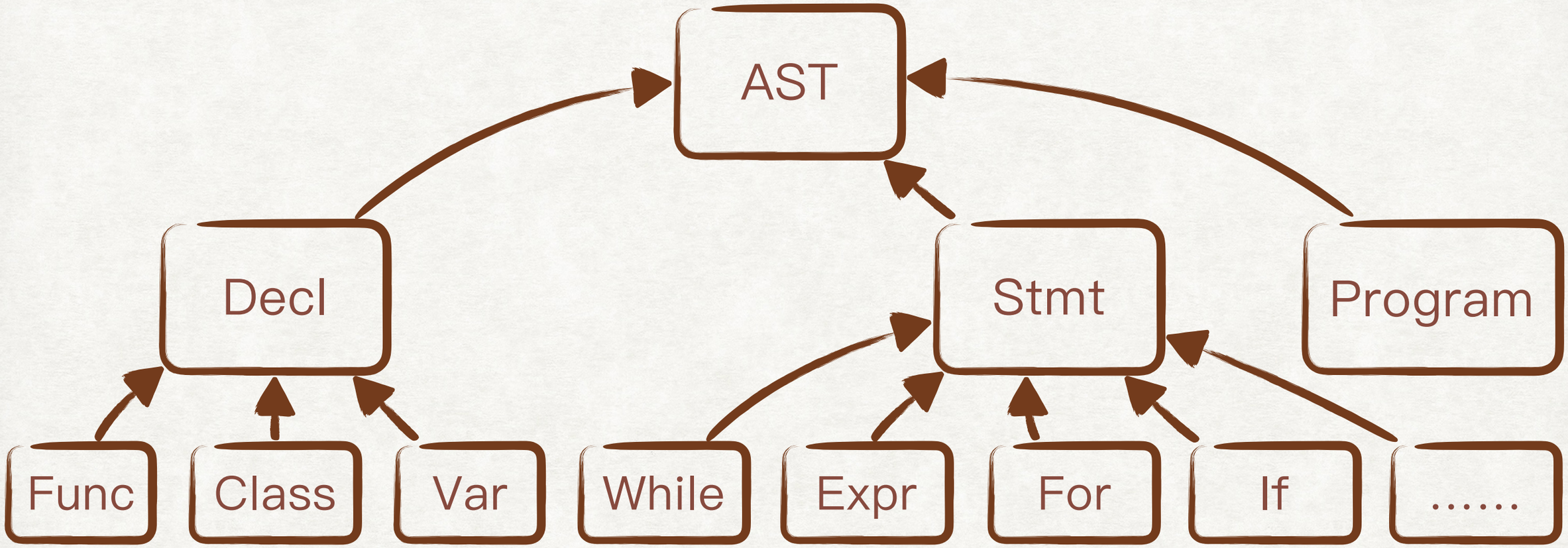
Structure: AST

A → B: A extends B



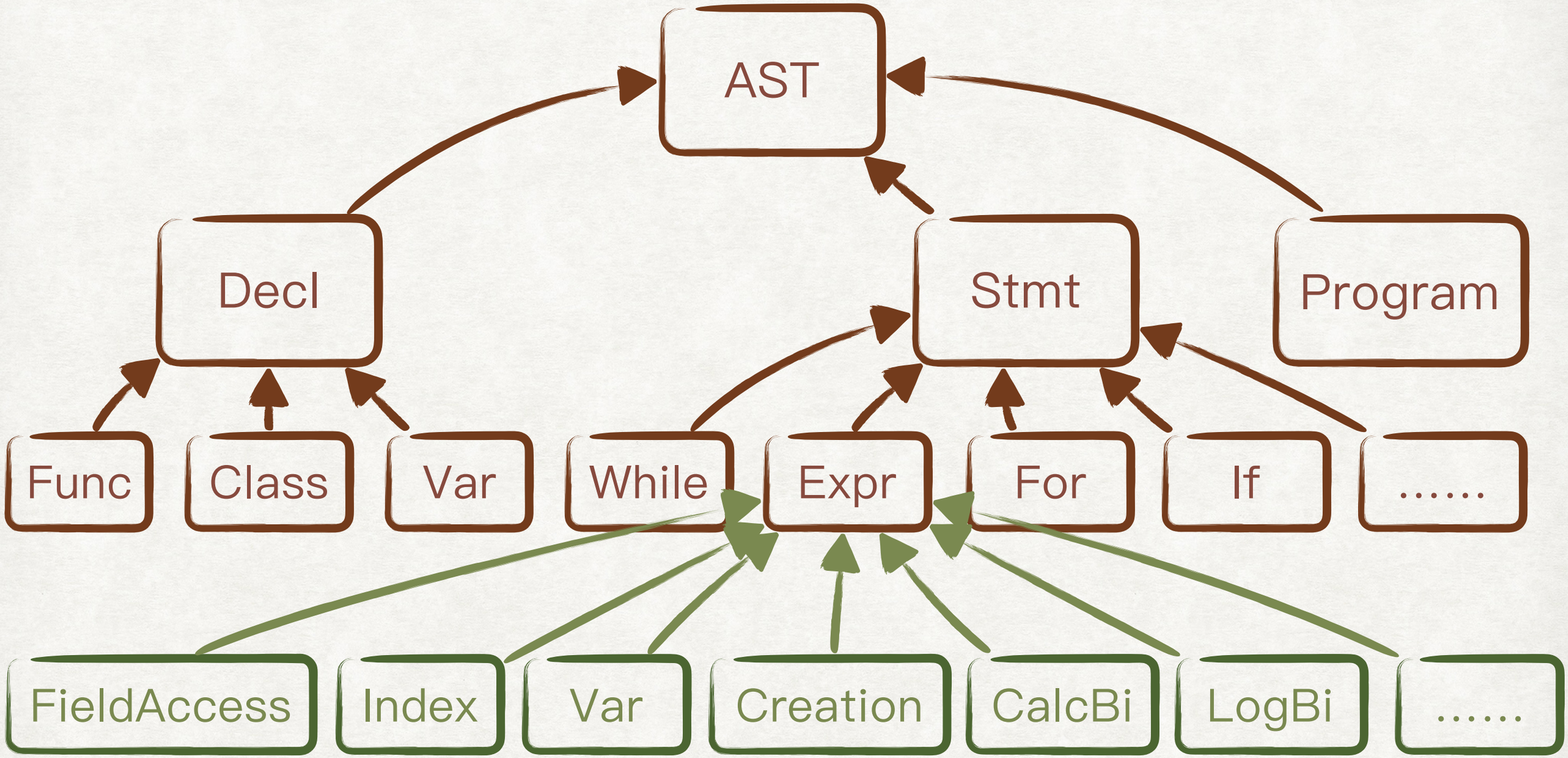
Structure: AST

A → B: A extends B



Structure: AST

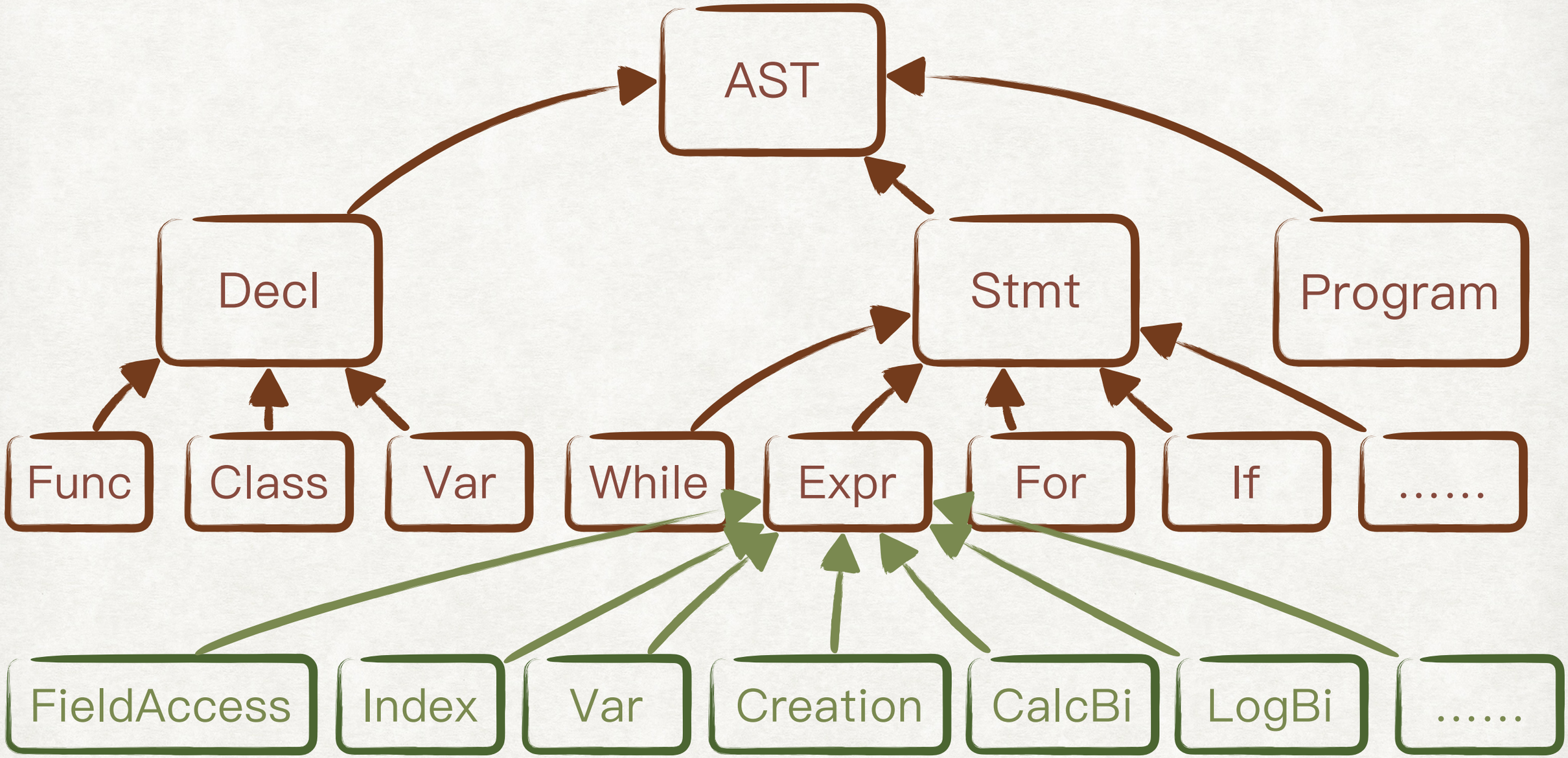
A → B: A extends B



Structure: AST

A ⟶ B: A extends B

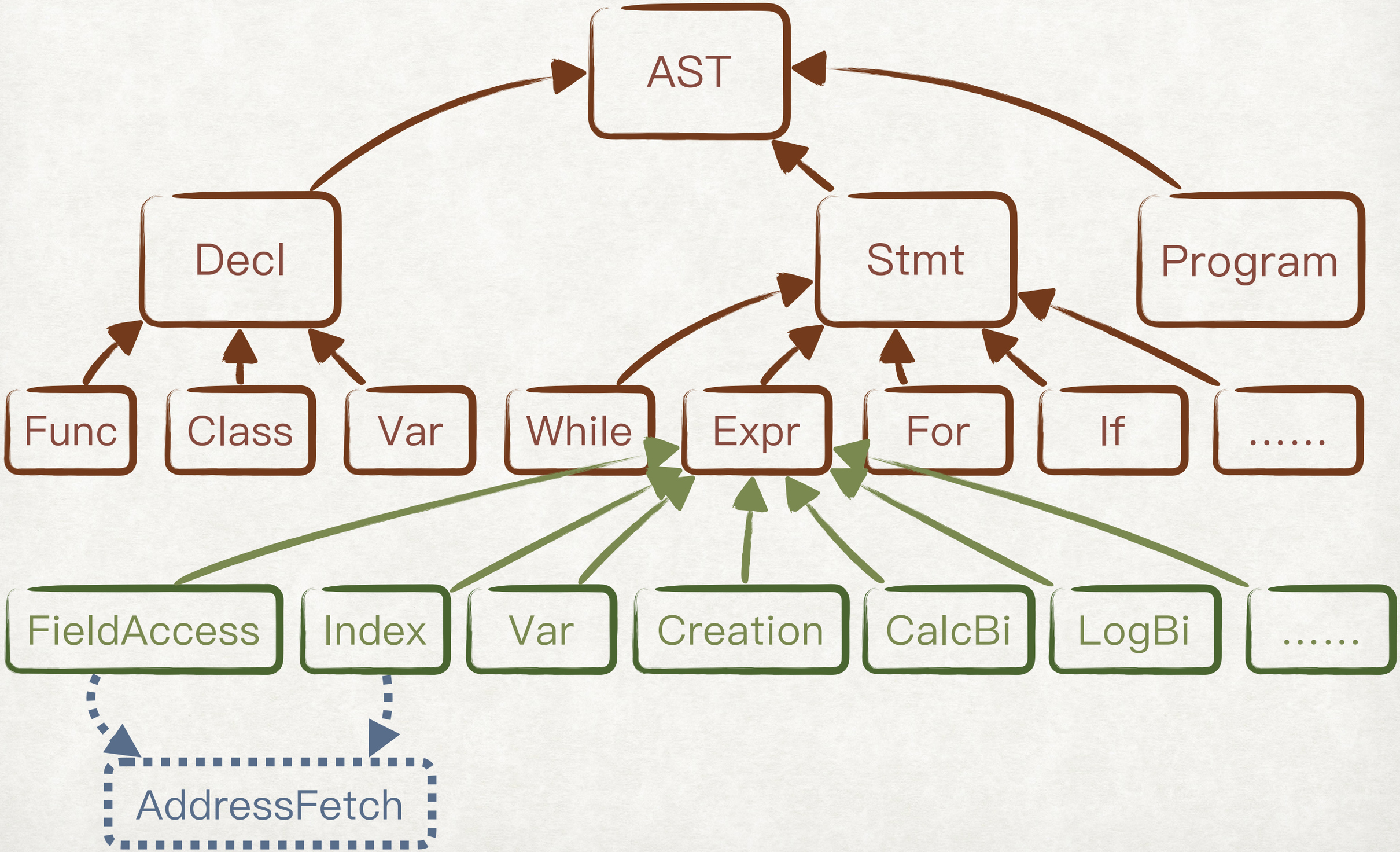
A ⇢ B: A implements B



Structure: AST

A ⟶ B: A extends B

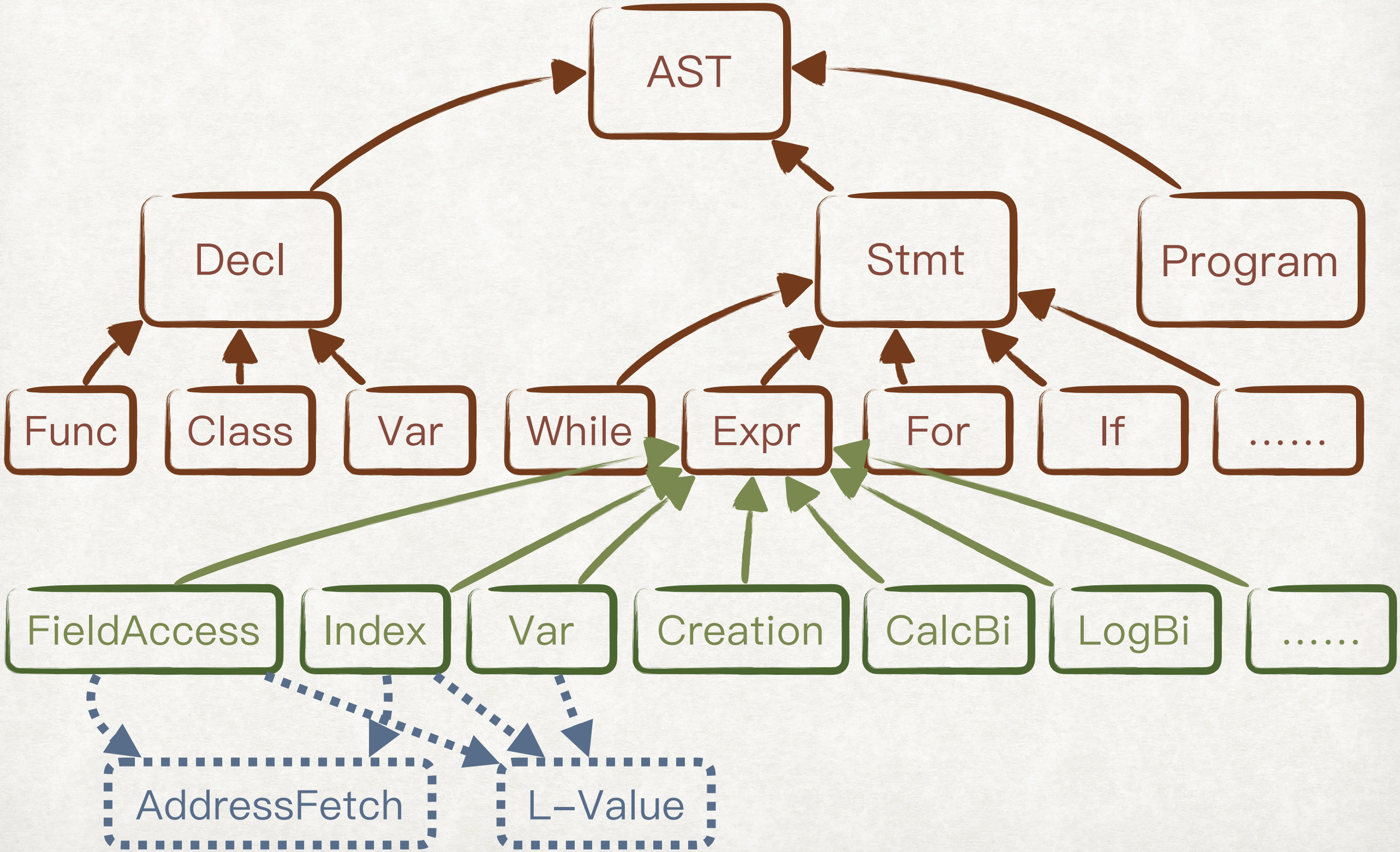
A ⇢ B: A implements B



Structure: AST

A ⟶ B: A extends B

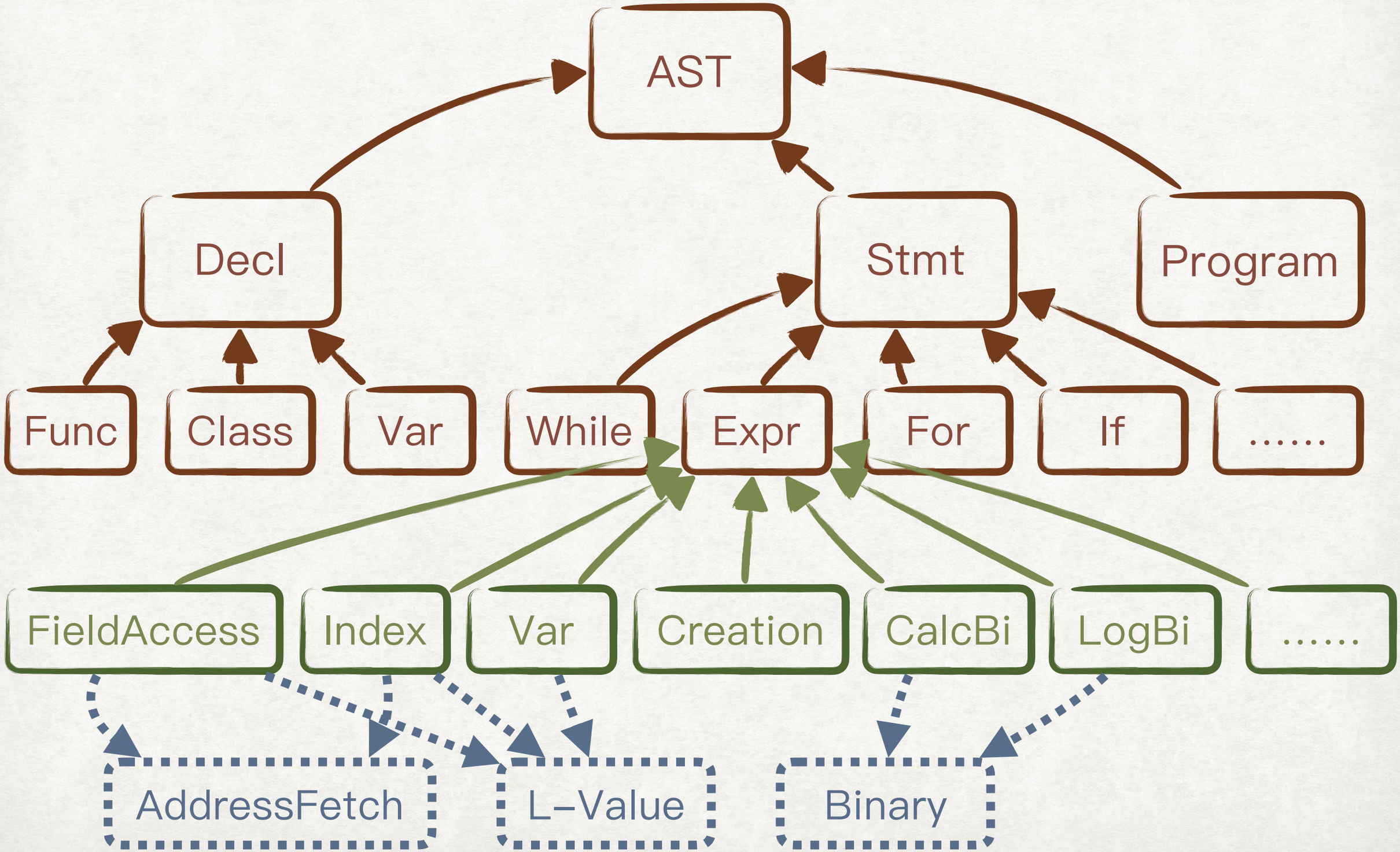
A ⇢ B: A implements B



Structure: AST

A ⟶ B: A extends B

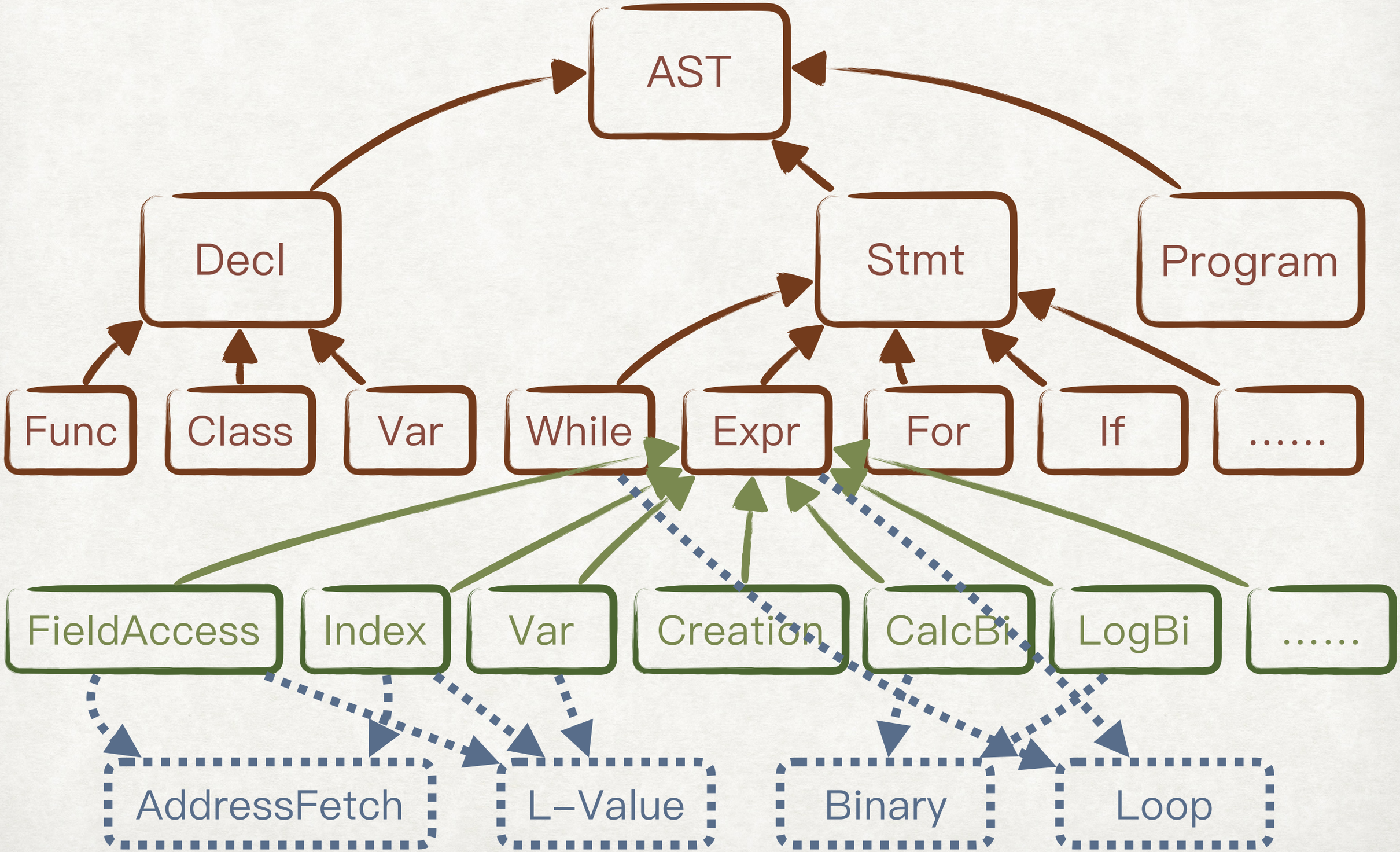
A ⇢ B: A implements B



Structure: AST

A ⟶ B: A extends B

A ⇢ B: A implements B



Structure: Symbol

Structure: Symbol

Symbol: Objects that

Structure: Symbol

Symbol: Objects that

1. store informations corresponding to specific Identifiers

Structure: Symbol

Symbol: Objects that

1. store <u>informations</u> corresponding to specific <u>Identifiers</u>
2. serve as Values in SymbolTable(a K-V map)

Structure: Symbol

Symbol: Objects that

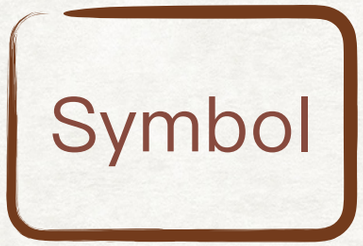
1. store <u>informations</u> corresponding to specific <u>Identifiers</u>
2. serve as Values in SymbolTable(a K-V map)

A ⟶ B: A extends B

Structure: Symbol

Symbol: Objects that

1. store <u>informations</u> corresponding to specific <u>Identifiers</u>
2. serve as Values in SymbolTable(a K-V map)

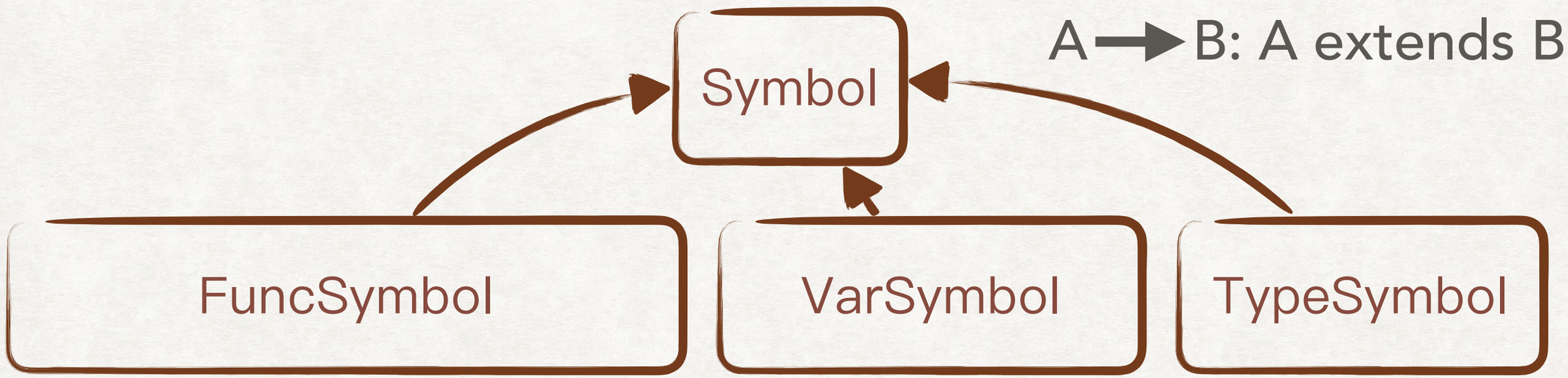


A → B: A extends B

Structure: Symbol

Symbol: Objects that

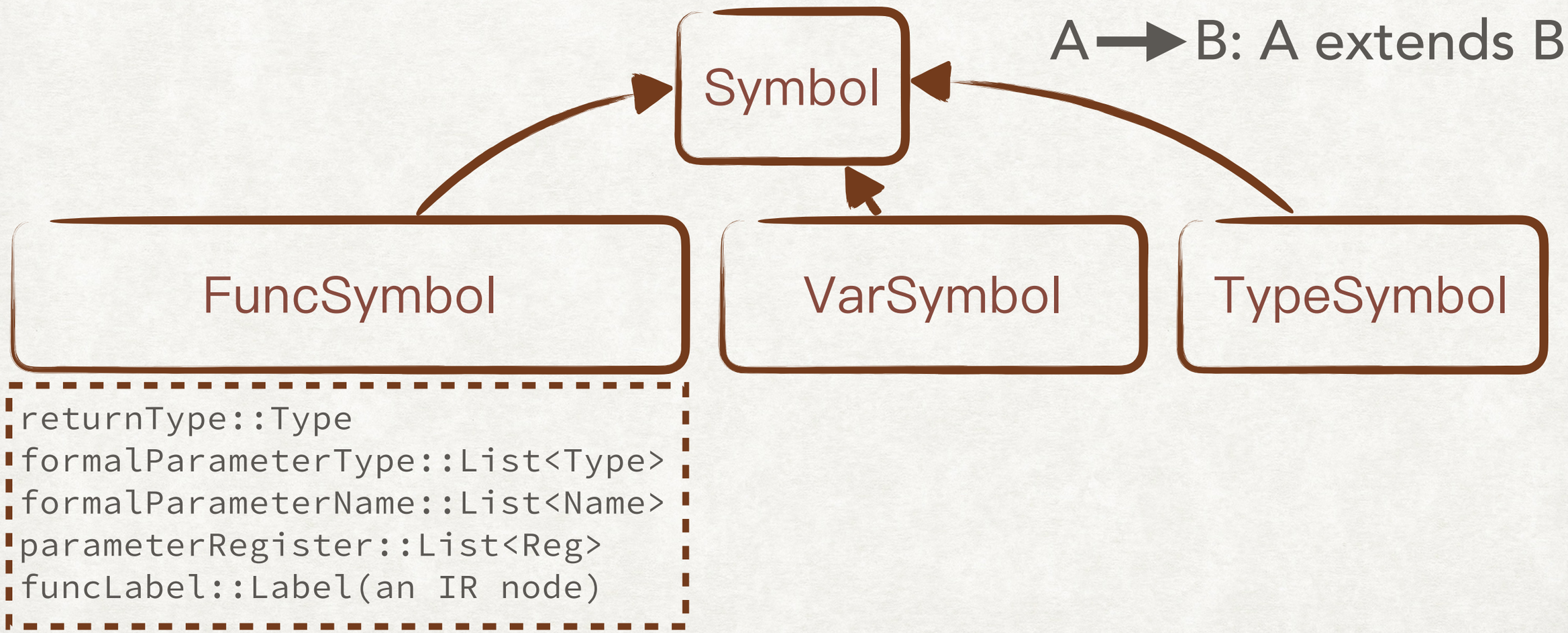
1. store <u>informations</u> corresponding to specific <u>Identifiers</u>
2. serve as Values in SymbolTable(a K-V map)



Structure: Symbol

Symbol: Objects that

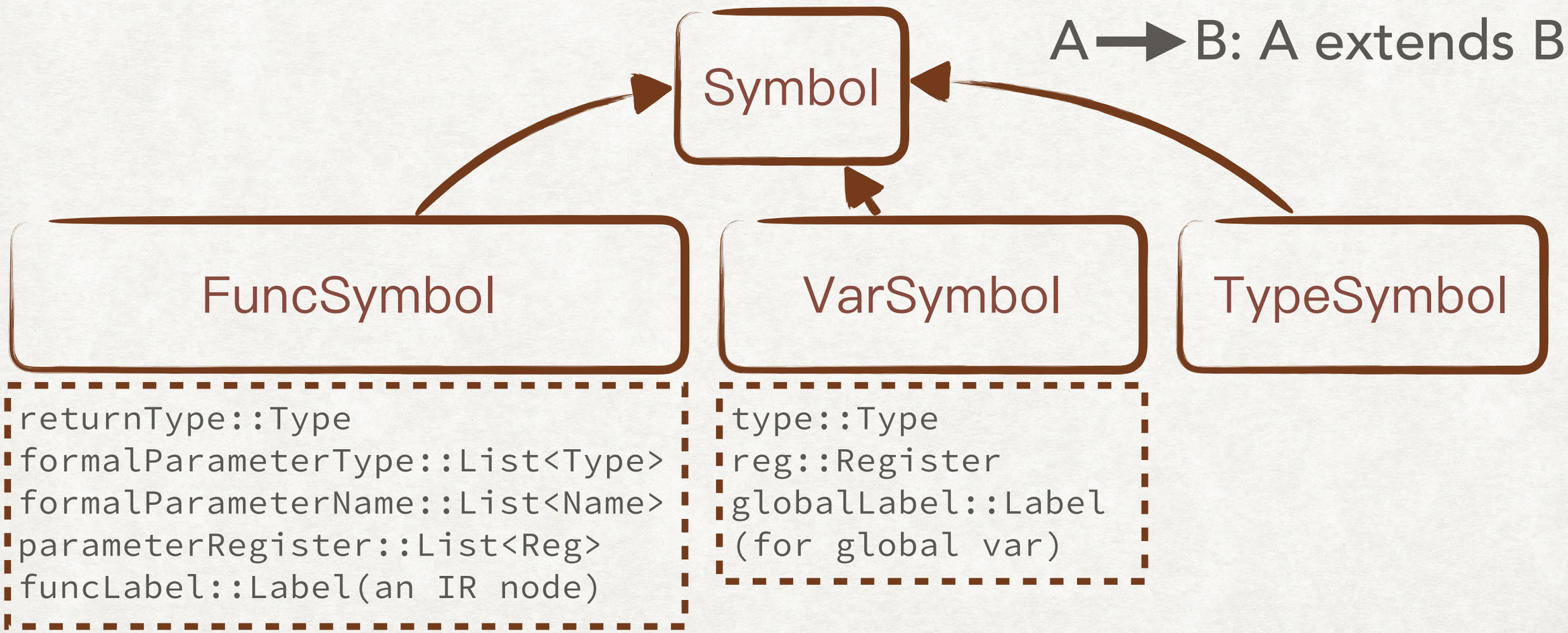
1. store <u>informations</u> corresponding to specific <u>Identifiers</u>
2. serve as Values in SymbolTable(a K-V map)



Structure: Symbol

Symbol: Objects that

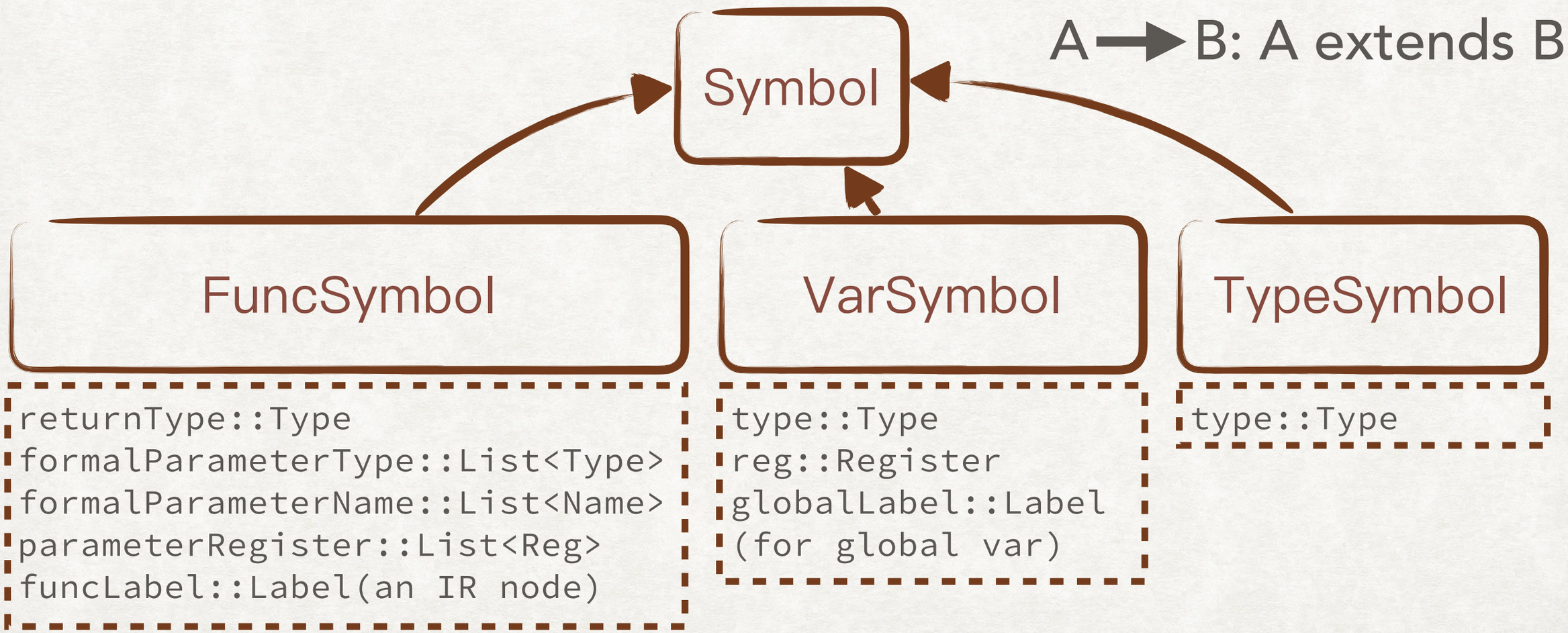
1. store <u>informations</u> corresponding to specific <u>Identifiers</u>
2. serve as Values in SymbolTable(a K-V map)



Structure: Symbol

Symbol: Objects that

1. store <u>informations</u> corresponding to specific <u>Identifiers</u>
2. serve as Values in SymbolTable(a K-V map)



Structure: Type

Structure: Type

Type: Objects that

Structure: Type

Type: Objects that

1. serve as distinguishers of different types of variables

Structure: Type

Type: Objects that

1. serve as distinguishers of different types of variables
2. store useful informations

Structure: Type

Type: Objects that

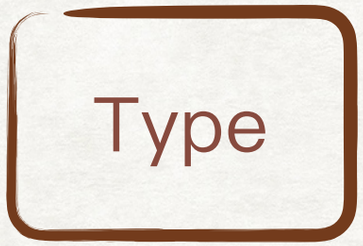
1. serve as distinguishers of different types of variables
2. store useful informations

A → B: A extends B

Structure: Type

Type: Objects that

1. serve as distinguishers of different types of variables
2. store useful informations

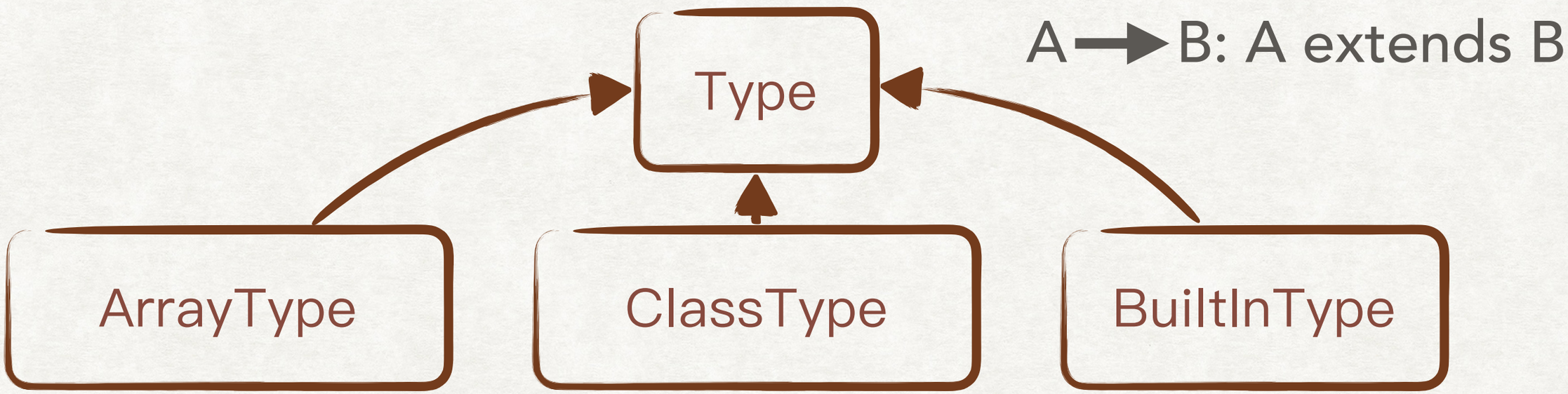


A → B: A extends B

Structure: Type

Type: Objects that

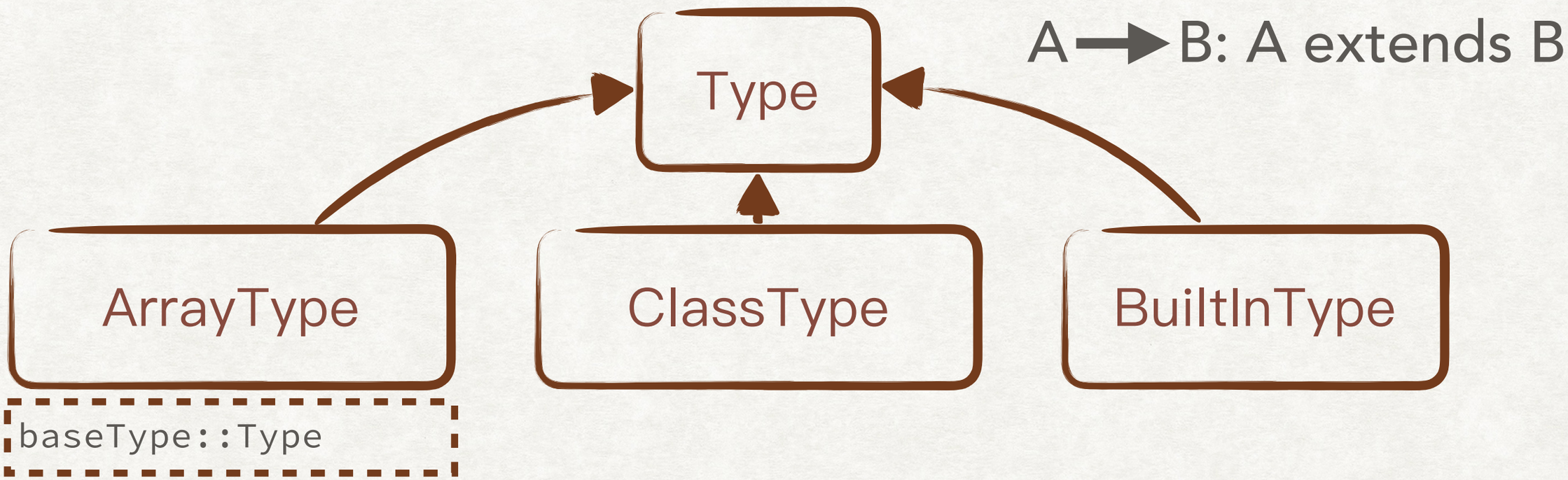
1. serve as distinguishers of different types of variables
2. store useful informations



Structure: Type

Type: Objects that

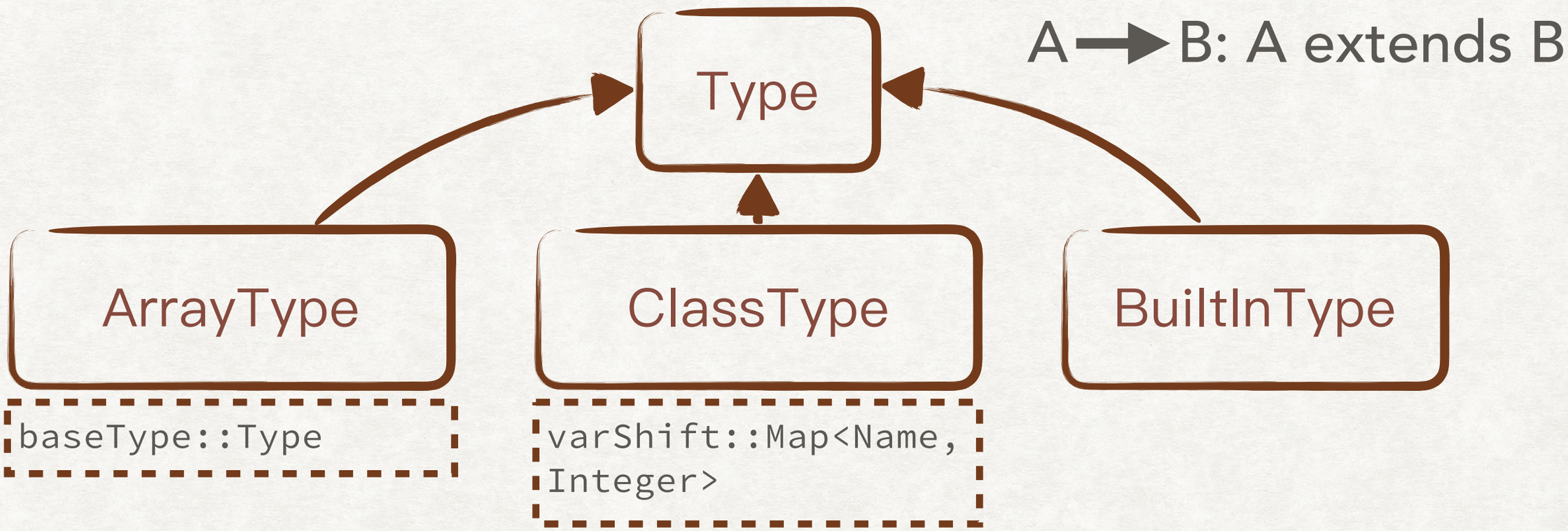
1. serve as distinguishers of different types of variables
2. store useful informations



Structure: Type

Type: Objects that

1. serve as distinguishers of different types of variables
2. store useful informations



Structure: Type

Structure: Type

The suit-check between Types:

Structure: Type

The suit-check between Types:

Is A suitable to be regarded as a B?

Structure: Type

The suit-check between Types:

Is A suitable to be regarded as a B?

1. Type A is a `ClassType` / `builtInType`

Structure: Type

The suit-check between Types:

Is A suitable to be regarded as a B?

1. Type A is a `ClassType` / `builtInType`

   `check A == B`

Structure: Type

The suit-check between Types:

Is A suitable to be regarded as a B?

1. Type A is a `ClassType` / `builtInType`

   `check A == B`

2. Type A is an `ArrayType`

Structure: Type

The suit-check between Types:

Is A suitable to be regarded as a B?

1. Type A is a `ClassType` / `builtInType`

   `check A == B`

2. Type A is an `ArrayType`

   `if (B == null) return true`

Structure: Type

The suit-check between Types:

Is A suitable to be regarded as a B?

1. Type A is a `ClassType` / `builtInType`

   ```
   check A == B
   ```

2. Type A is an `ArrayType`

   ```
   if (B == null) return true
   if (B != ArrayType) return false
   ```

Structure: Type

The suit-check between Types:

Is A suitable to be regarded as a B?

1. Type A is a `ClassType` / `builtInType`

   ```
   check A == B
   ```

2. Type A is an `ArrayType`

   ```
   if (B == null) return true
   if (B != ArrayType) return false
   check(A.baseType == B.baseType)
   ```

Structure: Type

The suit-check between Types:

Is A suitable to be regarded as a B?

1. Type A is a `ClassType` / `builtInType`

   ```
   check A == B
   ```

2. Type A is an `ArrayType`

   ```
   if (B == null) return true
   if (B != ArrayType) return false
   check(A.baseType == B.baseType)
   ```

Found it is the most simple way to handle:

Structure: Type

The suit-check between Types:

Is A suitable to be regarded as a B?

1. Type A is a `ClassType` / `builtInType`

   ```
   check A == B
   ```

2. Type A is an `ArrayType`

   ```
   if (B == null) return true
   if (B != ArrayType) return false
   check(A.baseType == B.baseType)
   ```

Found it is the most simple way to handle:

```
Foo: A[][][], bar: B[][]
```

Structure: Type

The suit-check between Types:

Is A suitable to be regarded as a B?

1. Type A is a `ClassType` / `builtInType`

   ```
   check A == B
   ```

2. Type A is an `ArrayType`

   ```
   if (B == null) return true
   if (B != ArrayType) return false
   check(A.baseType == B.baseType)
   ```

Found it is the most simple way to handle:

```
Foo: A[][][], bar: B[][]
foo.isSuitableAs(bar)?
```

Structure: SymbolTable

Structure: SymbolTable

SymbolTable: An Object that

Structure: SymbolTable

SymbolTable: An Object that

1. serves as a K-V map to collect information

Structure: SymbolTable

SymbolTable: An Object that

1. serves as a K-V map to collect information
2. capable of controlling scope

Structure: SymbolTable

SymbolTable: An Object that

1. serves as a K-V map to collect information
2. capable of controlling scope

- Fields: a series of built-In Type, functions

Structure: SymbolTable

SymbolTable: An Object that

1. serves as a K-V map to collect information
2. capable of controlling scope

- Fields: a series of built-In Type, functions predefine them in the map when initialized

Structure: SymbolTable

SymbolTable: An Object that

1. serves as a K-V map to collect information
2. capable of controlling scope

- Fields: a series of built-In Type, functions predefine them in the map when initialized
- Methods:

Structure: SymbolTable

SymbolTable: An Object that

1. serves as a K-V map to collect information
2. capable of controlling scope

- Fields: a series of built-In Type, functions predefine them in the map when initialized
- Methods:

```
void define(key, value)
```

Structure: SymbolTable

SymbolTable: An Object that

1. serves as a K-V map to collect information
2. capable of controlling scope

- Fields: a series of built-In Type, functions predefine them in the map when initialized
- Methods:

```
void define(key, value)
Symbol resolve(key)
```

Structure: SymbolTable

SymbolTable: An Object that

1. serves as a K-V map to collect information
2. capable of controlling scope

- Fields: a series of built-In Type, functions predefine them in the map when initialized
- Methods:

```
void define(key, value)
Symbol resolve(key)
void beginScope()
```

Structure: SymbolTable

SymbolTable: An Object that

1. serves as a K-V map to collect information
2. capable of controlling scope

- Fields: a series of built-In Type, functions predefine them in the map when initialized
- Methods:

```
void define(key, value)
Symbol resolve(key)
void beginScope()
void endScope()
```

Building AST

Building AST

- Use ANTLR's Listener and Visitor Pattern

Building AST

- Use ANTLR's Listener and Visitor Pattern
- For backward reference, we need three Passes:

Building AST

- Use ANTLR's Listener and Visitor Pattern
- For backward reference, we need three Passes:
  - First pass: (Listener) collect and check all class name

Building AST

- Use ANTLR's Listener and Visitor Pattern
- For backward reference, we need three Passes:
  - First pass: (Listener) collect and check all class name
  - Second pass: (Listener) collect and check all class field and function declarations

Building AST

- Use ANTLR's Listener and Visitor Pattern
- For backward reference, we need three Passes:
  - First pass: (Listener) collect and check all class name
  - Second pass: (Listener) collect and check all class field and function declarations
  - Third pass:

Building AST

- Use ANTLR's Listener and Visitor Pattern
- For backward reference, we need three Passes:
  - First pass: (Listener) collect and check all class name
  - Second pass: (Listener) collect and check all class field and function declarations
  - Third pass:
    - Type, function calling checking

Building AST

- Use ANTLR's Listener and Visitor Pattern
- For backward reference, we need three Passes:
  - First pass: (Listener) collect and check all class name
  - Second pass: (Listener) collect and check all class field and function declarations
  - Third pass:
    - Type, function calling checking
    - Building up the whole AST

Building AST

- Use ANTLR's Listener and Visitor Pattern
- For backward reference, we need three Passes:
  - First pass: (Listener) collect and check all class name
  - Second pass: (Listener) collect and check all class field and function declarations
  - Third pass:
    - Type, function calling checking
    - Building up the whole AST
- Lots of details when building AST, need good and coherent designs of AST, SymbolTable, Symbol and Type

Building AST

- Use ANTLR's Listener and Visitor Pattern
- For backward reference, we need three Passes:
  - First pass: (Listener) collect and check all class name
  - Second pass: (Listener) collect and check all class field and function declarations
  - Third pass:
    - Type, function calling checking
    - Building up the whole AST
- Lots of details when building AST, need good and coherent designs of AST, SymbolTable, Symbol and Type
- Java's exception handling provides great convenience

Building AST

Building AST

What needs to be store in the AST node?

Building AST

What needs to be store in the AST node?

- position of the corresponding code segment

Building AST

What needs to be store in the AST node?

- position of the corresponding code segment
- `Expr` node, for every node of this kind represents a computed value:

Building AST

What needs to be store in the AST node?

- position of the corresponding code segment
- `Expr` node, for every node of this kind represents a computed value:
  - Type

Building AST

What needs to be store in the AST node?

- position of the corresponding code segment
- `Expr` node, for every node of this kind represents a computed value:
  - Type
  - Operand (For the convenience of generating IR)

Building AST

What needs to be store in the AST node?

- position of the corresponding code segment
- `Expr` node, for every node of this kind represents a computed value:
  - Type
  - Operand (For the convenience of generating IR)
- `VarExpr` node, for it represents a leaf node in AST

Building AST

What needs to be store in the AST node?

- position of the corresponding code segment
- `Expr` node, for every node of this kind represents a computed value:
  - Type
  - Operand (For the convenience of generating IR)
- `VarExpr` node, for it represents a leaf node in AST
  - VarSymbol

Building AST

What needs to be store in the AST node?

- position of the corresponding code segment
- `Expr` node, for every node of this kind represents a computed value:
  - Type
  - Operand (For the convenience of generating IR)
- `VarExpr` node, for it represents a leaf node in AST
  - VarSymbol

We don't need SymbolTable after building AST, for all useful infos have been stored in AST node!!

Building AST

Building AST

Why do I store infos in AST?

Building AST

Why do I store infos in AST?

```
int a = 1;
int main(){
    int a = a + 1;
    print(toString(a));
}
```

Building AST

Why do I store infos in AST?

```
int a = 1;
int main(){
    int a = a + 1;
    print(toString(a));
}
```

If we don's store infos in AST, we'll have to rebuild the SymbolTable when we want to walk through AST to gather more infos!!

Building AST

Building AST

Tricks:

Building AST

Tricks:

- transform string op to function call, e.g.

Building AST

Tricks:

- transform string op to function call, e.g.
  - string A + string B => stringConcatenate(A, B)

Building AST

Tricks:

- transform string op to function call, e.g.
    - string A + string B => stringConcatenate(A, B)
    - string A < string B => stringLess(A, B)

Building AST

Tricks:

- transform string op to function call, e.g.
    - string A + string B => stringConcatenate(A, B)
    - string A < string B => stringLess(A, B)
- constant combination, e.g.

Building AST

Tricks:

- transform string op to function call, e.g.
  - string A + string B => stringConcatenate(A, B)
  - string A < string B => stringLess(A, B)
- constant combination, e.g.
  - BinaryAdd(1, 1) => IntExpr(2)

Building AST

Building AST

Friendly Error report: e.g.

Building AST

Friendly Error report: e.g.

`final/compile_error/asiop-1-5100379071-puyouer.mx`

Building AST

Friendly Error report: e.g.

```
final/compile_error/asiop-1-5100379071-puyouer.mx
line 2: Wrong assignment operation on non-Lvalue <“a”>
```

Building AST

Friendly Error report: e.g.

```
final/compile_error/asiop-1-5100379071-puyouer.mx
line 2: Wrong assignment operation on non-Lvalue <“a”>
final/compile_error/asiop-7-5120309049-liaochao.mx
```

Building AST

Friendly Error report: e.g.

```
final/compile_error/asiop-1-5100379071-puyouer.mx
line 2: Wrong assignment operation on non-Lvalue <“a”>
final/compile_error/asiop-7-5120309049-liaochao.mx
line 2: Unsuitable type in variable declaration, expect <string> but meet <null>
```

Building AST

Friendly Error report: e.g.

```
final/compile_error/asiop-1-5100379071-puyouer.mx
line 2: Wrong assignment operation on non-Lvalue <“a”>
final/compile_error/asiop-7-5120309049-liaochao.mx
line 2: Unsuitable type in variable declaration, expect <string> but meet <null>
final/compile_error/func-4-5100379071-puyouer.mx
```

Building AST

Friendly Error report: e.g.

```
final/compile_error/asiop-1-5100379071-puyouer.mx
line 2: Wrong assignment operation on non-Lvalue <“a”>
final/compile_error/asiop-7-5120309049-liaochao.mx
line 2: Unsuitable type in variable declaration, expect <string> but meet <null>
final/compile_error/func-4-5100379071-puyouer.mx
line 7: Function call parameter mismatching, see function declaration at <f>
```

Structure: IR

Structure: IR

Why do we need IR?

Structure: IR

Why do we need IR?

A step further to lower the abstraction level!

Structure: IR

Why do we need IR?

A step further to lower the abstraction level!

- Branch
- Jump
- Return
- Label
- Binary
- LoadImm
- Move
- Call
- Neg
- New
- Not
- Load
- Load Parameter
- Load Address
- Load FromLabel
- Store
- StoreLabel

Structure: IR

Structure: IR

Consideration:

Structure: IR

Consideration:

- Use lower-level instructions to represent the AST

Structure: IR

Consideration:

- Use lower-level instructions to represent the AST
- No need for tree structure — do no more thing than AST

Structure: IR

Consideration:

- Use lower-level instructions to represent the AST
- No need for tree structure — do no more thing than AST
- Resemble real MIPS, but use infinite registers

Structure: IR

Consideration:

- Use lower-level instructions to represent the AST
- No need for tree structure — do no more thing than AST
- Resemble real MIPS, but use infinite registers
- special notice on:

Structure: IR

Consideration:

- Use lower-level instructions to represent the AST
- No need for tree structure — do no more thing than AST
- Resemble real MIPS, but use infinite registers
- special notice on:
    1. String constant

Structure: IR

Consideration:

- Use lower-level instructions to represent the AST
- No need for tree structure — do no more thing than AST
- Resemble real MIPS, but use infinite registers
- special notice on:
    1. String constant
    2. Function parameter

Structure: IR

Consideration:

- Use lower-level instructions to represent the AST
- No need for tree structure — do no more thing than AST
- Resemble real MIPS, but use infinite registers
- special notice on:
    1. String constant
    2. Function parameter
    3. Global Variable

Structure: IR

Consideration:

- Use lower-level instructions to represent the AST
- No need for tree structure — do no more thing than AST
- Resemble real MIPS, but use infinite registers
- special notice on:
    1. String constant
    2. Function parameter
    3. Global Variable
    4. Array

Structure: IR

Consideration:

- Use lower-level instructions to represent the AST
- No need for tree structure — do no more thing than AST
- Resemble real MIPS, but use infinite registers
- special notice on:
    1. String constant
    2. Function parameter
    3. Global Variable
    4. Array
    5. Loop

Structure: IR

Consideration:

- Use lower-level instructions to represent the AST
- No need for tree structure — do no more thing than AST
- Resemble real MIPS, but use infinite registers
- special notice on:
    1. String constant
    2. Function parameter
    3. Global Variable
    4. Array
    5. Loop
    6. If statement

Structure: IR

Consideration:

- Use lower-level instructions to represent the AST
- No need for tree structure — do no more thing than AST
- Resemble real MIPS, but use infinite registers
- special notice on:
    1. String constant
    2. Function parameter
    3. Global Variable
    4. Array
    5. Loop
    6. If statement
    7. Short-cut evaluation

Structure: IR

Structure: IR

String Constant:

Structure: IR

String Constant:

MIPS:

```
.data
    .word 4
str:.asciiz “haha”
    .align 2
.text
    ……
    la $a0, str
    li $v0, 4
    syscall
    ……
```

Structure: IR

String Constant:

MIPS:

```
.data
    .word 4
str:.asciiz “haha”
    .align 2
.text
    ……
    la $a0, str
    li $v0, 4
    syscall
    ……
```

We need the `la` to support string constant!

Structure: IR

String Constant:

MIPS:

```
.data
    .word 4
str:.asciiz “haha”
    .align 2
.text
    ……
    la $a0, str
    li $v0, 4
    syscall
    ……
```

We need the `la` to support string constant!

Then there is a `LoadAddress` in my IR

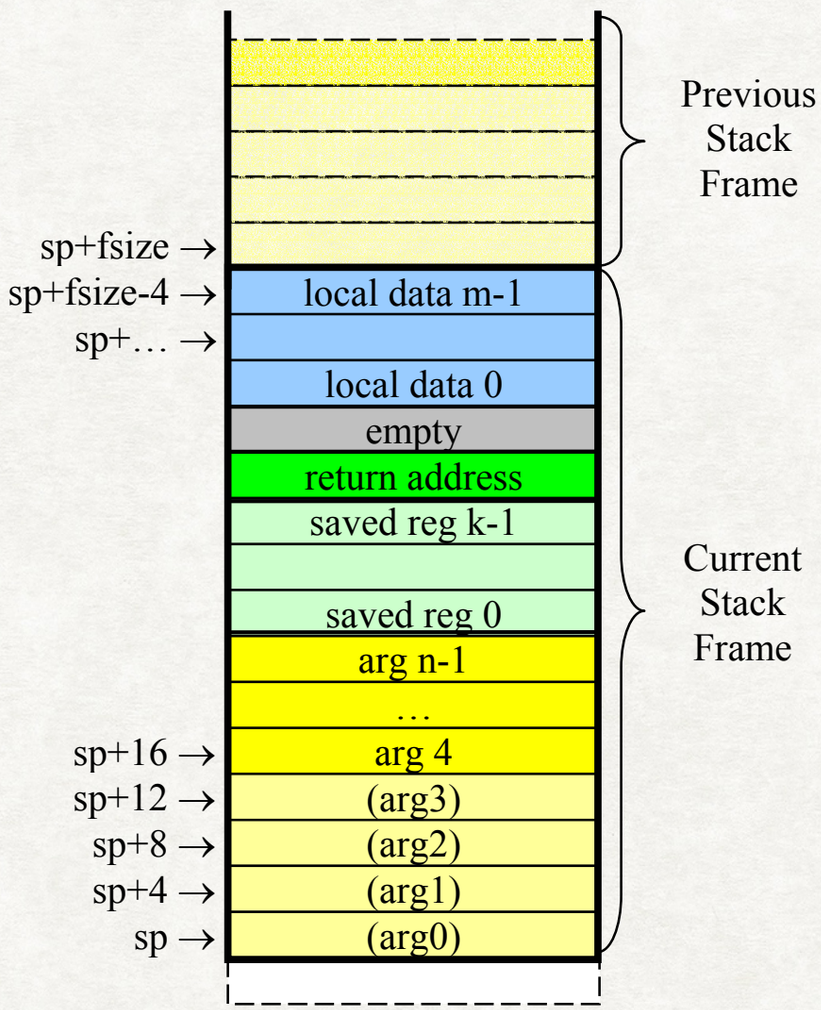
Structure: IR

Structure: IR

Function parameters

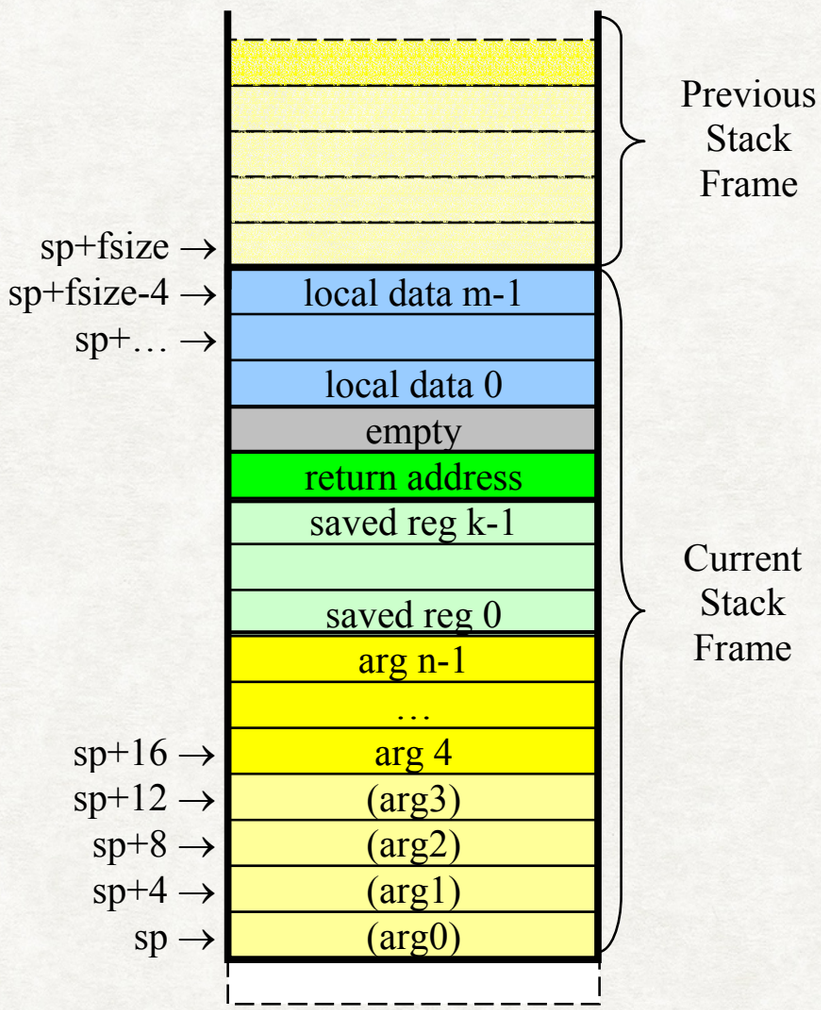
Structure: IR

Function parameters



Structure: IR

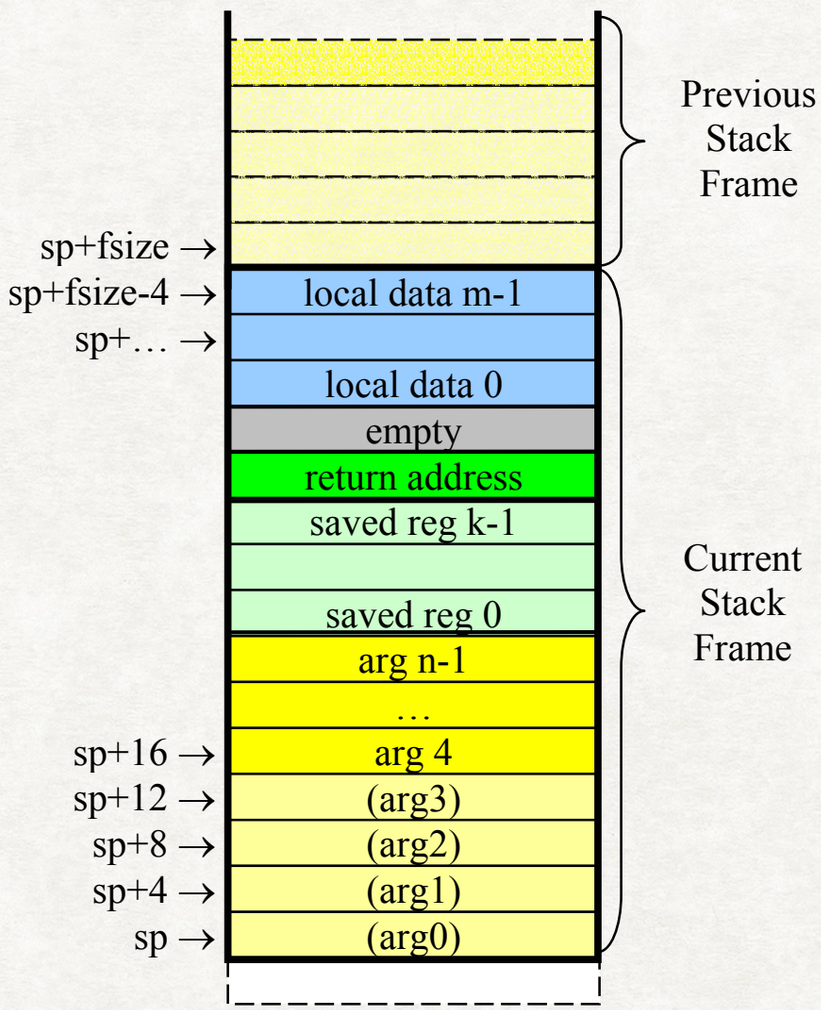
Function parameters



Parameters are stored in stack, but we now know nothing about how large the stack is!

Structure: IR

Function parameters

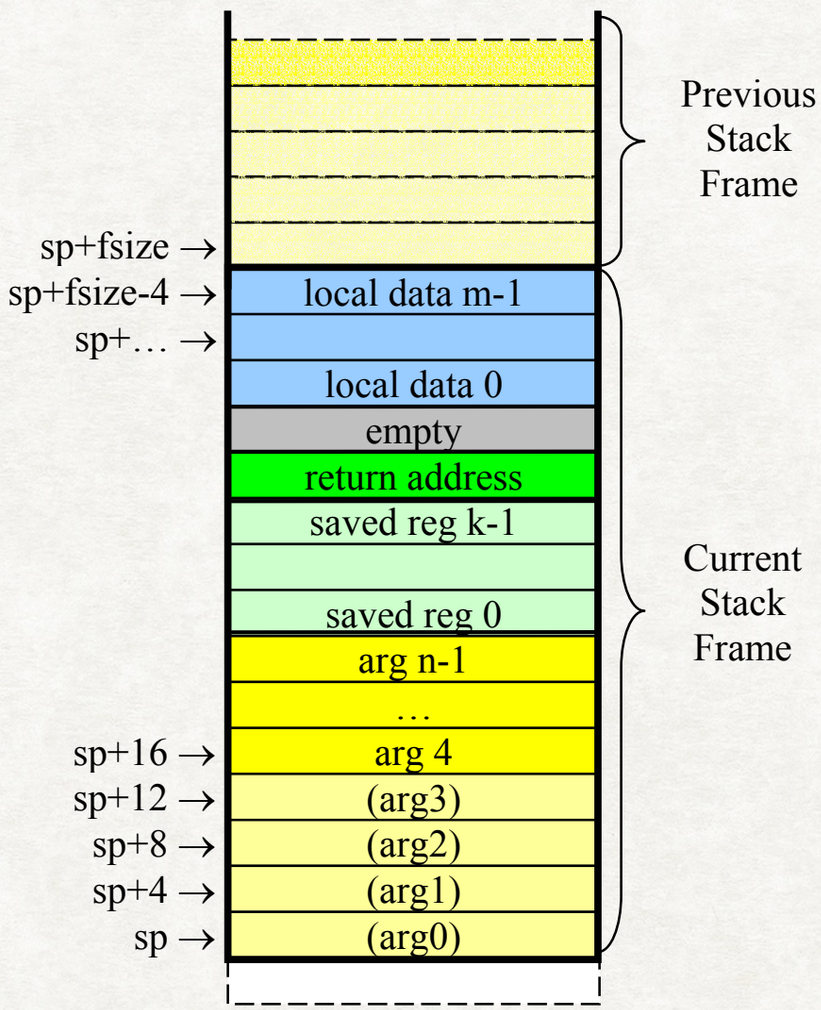


Parameters are stored in stack, but we now know nothing about how large the stack is!

We need a place holder in IR!

Structure: IR

Function parameters



Parameters are stored in stack, but we now know nothing about how large the stack is!

We need a place holder in IR!

Then there is a `LoadParameter` in my IR

Structure: IR

Structure: IR

Global Variable

Structure: IR

Global Variable

```
int a = 0;
int main(){
    a = a + 1;
}
```

Structure: IR

Global Variable

```
int a = 0;
int main(){
    a = a + 1;
}
```

MIPS:

```
.data
var_a:.word 0
.text
    ......
    lw  $t0, var_a
    add $t0, $t0, 1
    sw  $t0, var_a
    ......
```

Structure: IR

Global Variable

```
int a = 0;
int main(){
    a = a + 1;
}
```

MIPS:

```
.data
var_a:.word 0
.text
    ……
    lw  $t0, var_a
    add $t0, $t0, 1
    sw  $t0, var_a
    ……
```

We need `LoadLabel` and `StoreLabel` to support global variable!

Structure: IR

Structure: IR

Array

Structure: IR

Array

```
class A{
    int a;
}
int main(){
    A[] test = new A[10];
    test[0] = new A;
    test[0].a = 1;
}
```

Structure: IR

Array

```
class A{
    int a;
}
int main(){
    A[] test = new A[10];
    test[0] = new A;
    test[0].a = 1;
}
```

- store size of array in front of the initial place
- count the shift!

Structure: IR

Structure: IR

Loop

Structure: IR

Loop

```
for (init; cond; increment)
    body
```

Structure: IR

Loop

```
for (init; cond; increment)
    body
```

```
init
branch cond T=for_begin F=for_end
Label:for_begin
body
increment
branch cond T=for_begin F=for_end
Label:for_end
```

Structure: IR

Structure: IR

Short-cut evaluation

Structure: IR

Short-cut evaluation

```
void buildCondition(T, F, condition)
    Label temp
    if (condition is LogAnd)
        buildCondition(temp, F, condition.lhs)
        place(temp)
        buildCondition(T, F, condition.lhs)
    else if (condition is LogOr)
        buildCondition(T, temp, condition.lhs)
        place(temp)
        buildCondition(T, F, condition.lhs)
    ……
```

Structure: IR

Short-cut evaluation

```
void buildCondition(T, F, condition)
    Label temp
    if (condition is LogAnd)
        buildCondition(temp, F, condition.lhs)
        place(temp)
        buildCondition(T, F, condition.lhs)
    else if (condition is LogOr)
        buildCondition(T, temp, condition.lhs)
        place(temp)
        buildCondition(T, F, condition.lhs)
    ……
```

left-associative log evaluation, simple & extremely efficient:

Structure: IR

Short-cut evaluation

```
void buildCondition(T, F, condition)
    Label temp
    if (condition is LogAnd)
        buildCondition(temp, F, condition.lhs)
        place(temp)
        buildCondition(T, F, condition.lhs)
    else if (condition is LogOr)
        buildCondition(T, temp, condition.lhs)
        place(temp)
        buildCondition(T, F, condition.lhs)
    ……
```

left-associative log evaluation, simple & extremely efficient:

superloop-5090379042-jiaxiao.mx

Structure: IR

Short-cut evaluation

```
void buildCondition(T, F, condition)
    Label temp
    if (condition is LogAnd)
        buildCondition(temp, F, condition.lhs)
        place(temp)
        buildCondition(T, F, condition.lhs)
    else if (condition is LogOr)
        buildCondition(T, temp, condition.lhs)
        place(temp)
        buildCondition(T, F, condition.lhs)
    ……
```

left-associative log evaluation, simple & extremely efficient:

```
superloop-5090379042-jiaxiao.mx
Threshold:5000000
```

Structure: IR

Short-cut evaluation

```
void buildCondition(T, F, condition)
    Label temp
    if (condition is LogAnd)
        buildCondition(temp, F, condition.lhs)
        place(temp)
        buildCondition(T, F, condition.lhs)
    else if (condition is LogOr)
        buildCondition(T, temp, condition.lhs)
        place(temp)
        buildCondition(T, F, condition.lhs)
    ……
```

left-associative log evaluation, simple & extremely efficient:

```
superloop-5090379042-jiaxiao.mx
Threshold:5000000
NaiveMIPS:4801078
```

Structure: IR

Short-cut evaluation

```
void buildCondition(T, F, condition)
    Label temp
    if (condition is LogAnd)
        buildCondition(temp, F, condition.lhs)
        place(temp)
        buildCondition(T, F, condition.lhs)
    else if (condition is LogOr)
        buildCondition(T, temp, condition.lhs)
        place(temp)
        buildCondition(T, F, condition.lhs)
    ……
```

left-associative log evaluation, simple & extremely efficient:

```
superloop-5090379042-jiaxiao.mx
Threshold:5000000
NaiveMIPS:4801078
FinalMIPS: 976129
```

Structure: IR

Structure: IR

Experience on IR building:

Structure: IR

Experience on IR building:

1. we have to have a deep understanding on what kind of IR do we want to have when we designing IR

Structure: IR

Experience on IR building:

1. we have to have a deep understanding on what kind of IR do we want to have when we designing IR
2. It will be much better to have a clear idea on what object code looks like before building IR

Structure: IR

Experience on IR building:

1. we have to have a deep understanding on what kind of IR do we want to have when we designing IR
2. It will be much better to have a clear idea on what object code looks like before building IR
3. pay special notice on operations that have side-effect: assign, selfIncrement and selfDecrement

Structure: CFG

Structure: CFG

Why do we want to have CFGs?

Structure: CFG

Why do we want to have CFGs?

for the convenience of Liveness Analysis

Structure: CFG

Why do we want to have CFGs?

for the convenience of Liveness Analysis

CFG:

Structure: CFG

Why do we want to have CFGs?

for the convenience of Liveness Analysis

CFG:

1. a function is a CFG

Structure: CFG

Why do we want to have CFGs?

for the convenience of Liveness Analysis

CFG:

1. a function is a CFG
2. a CFG consists of Basic Blocks

Structure: CFG

Why do we want to have CFGs?

for the convenience of Liveness Analysis

CFG:

1. a function is a CFG
2. a CFG consists of Basic Blocks
3. A Basic Block is just a series of IRNode, just have to guarantee that this series begins with `Label`, ends with `return/jump/branch`, and no other `return/jump/branch` in the IRNode series.

Structure: CFG

Why do we want to have CFGs?

for the convenience of Liveness Analysis

CFG:

1. a function is a CFG
2. a CFG consists of Basic Blocks
3. A Basic Block is just a series of IRNode, just have to guarantee that this series begins with `Label`, ends with `return/jump/branch`, and no other `return/jump/branch` in the IRNode series.

Very very simple to build!

Liveness Analysis

Liveness Analysis

Why do we want to do Liveness Analysis?

Liveness Analysis

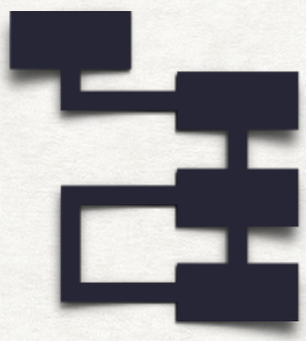
Why do we want to do Liveness Analysis?

- we want to know how long a variable will live to allocate regs

Liveness Analysis

Why do we want to do Liveness Analysis?

we want to know how long a variable will live to allocate regs

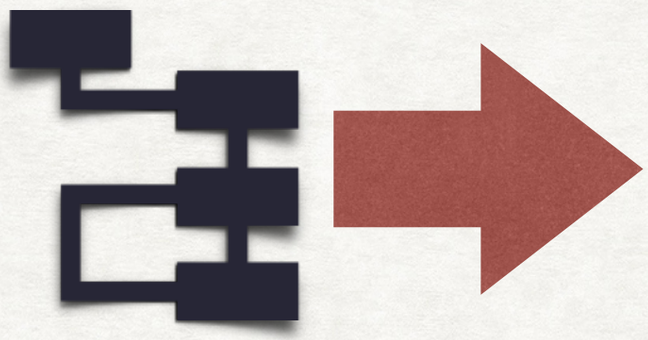


CFG

Liveness Analysis

Why do we want to do Liveness Analysis?

we want to know how long a variable will live to allocate regs

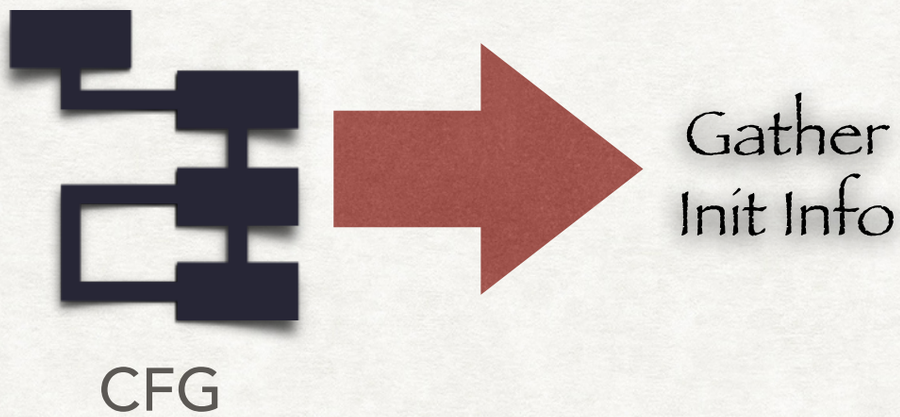


CFG

Liveness Analysis

Why do we want to do Liveness Analysis?

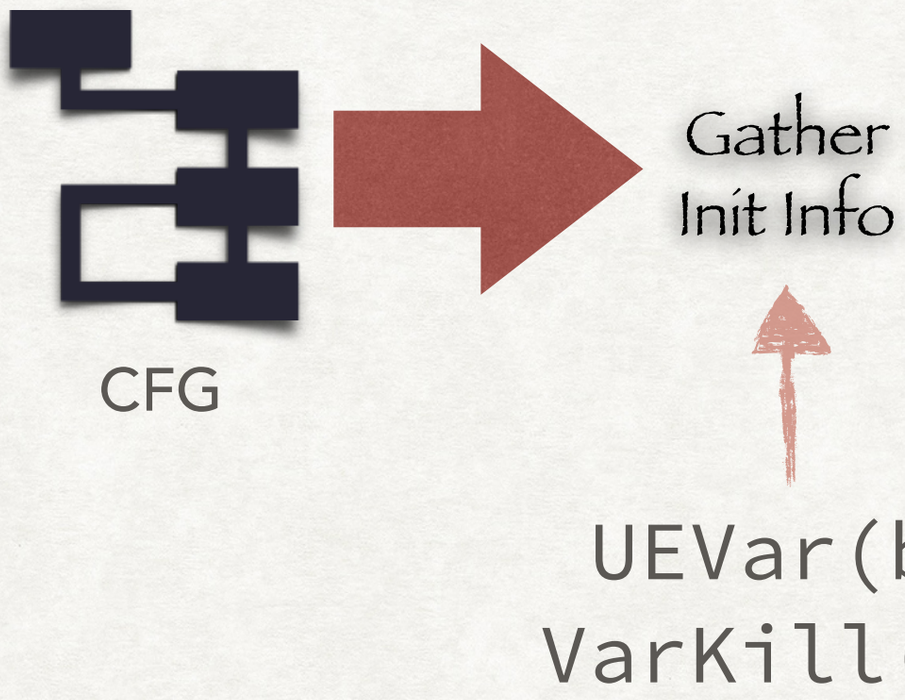
we want to know how long a variable will live to allocate regs



Liveness Analysis

Why do we want to do Liveness Analysis?

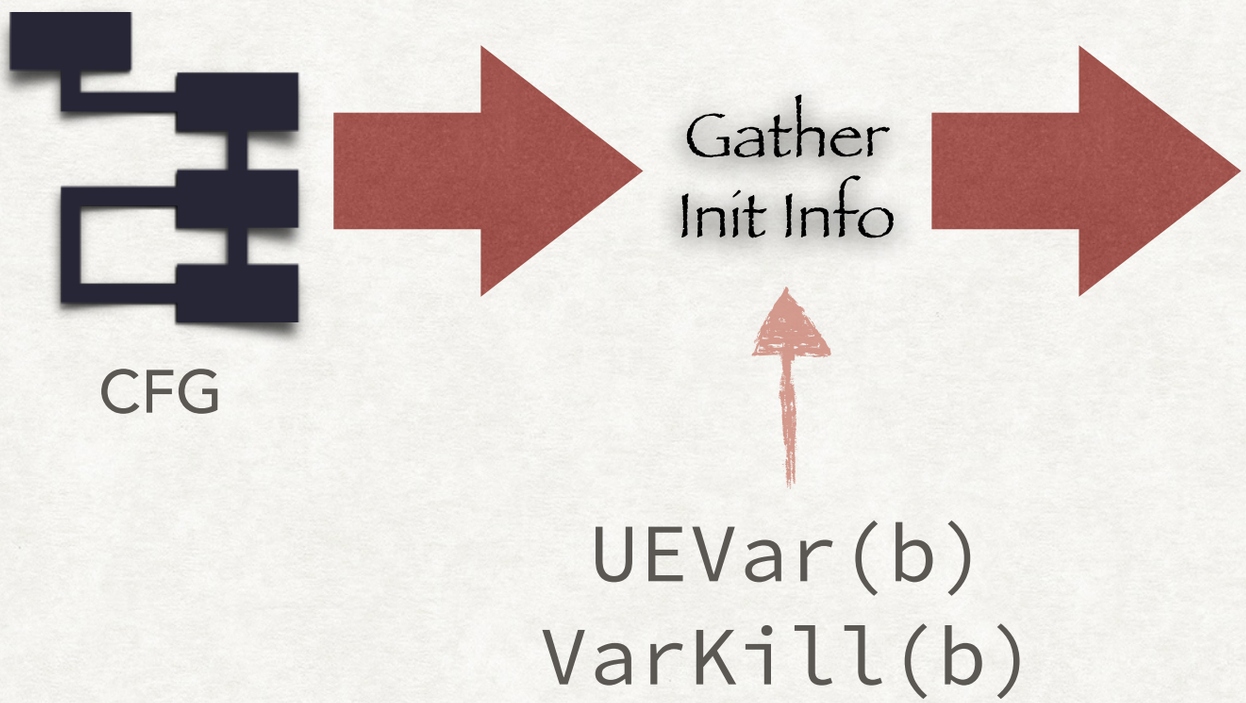
we want to know how long a variable will live to allocate regs



Liveness Analysis

Why do we want to do Liveness Analysis?

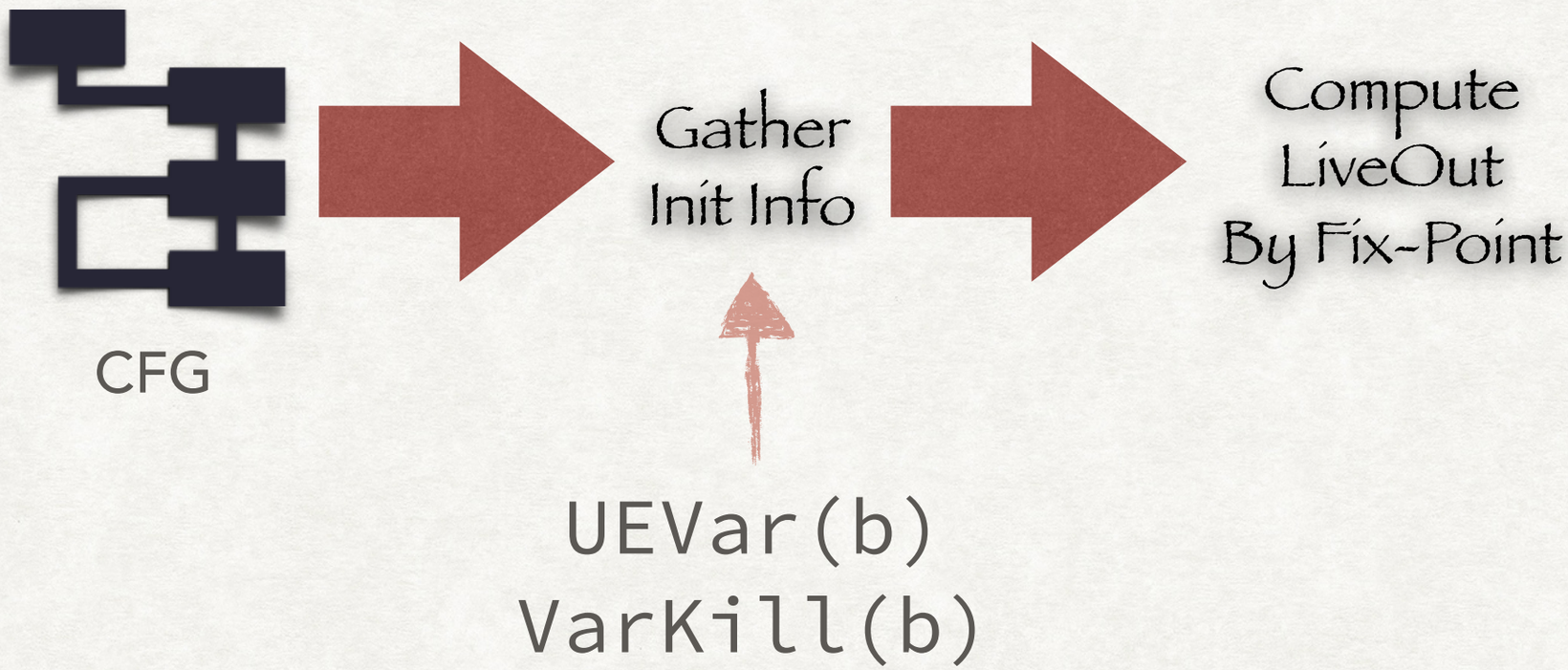
we want to know how long a variable will live to allocate regs



Liveness Analysis

Why do we want to do Liveness Analysis?

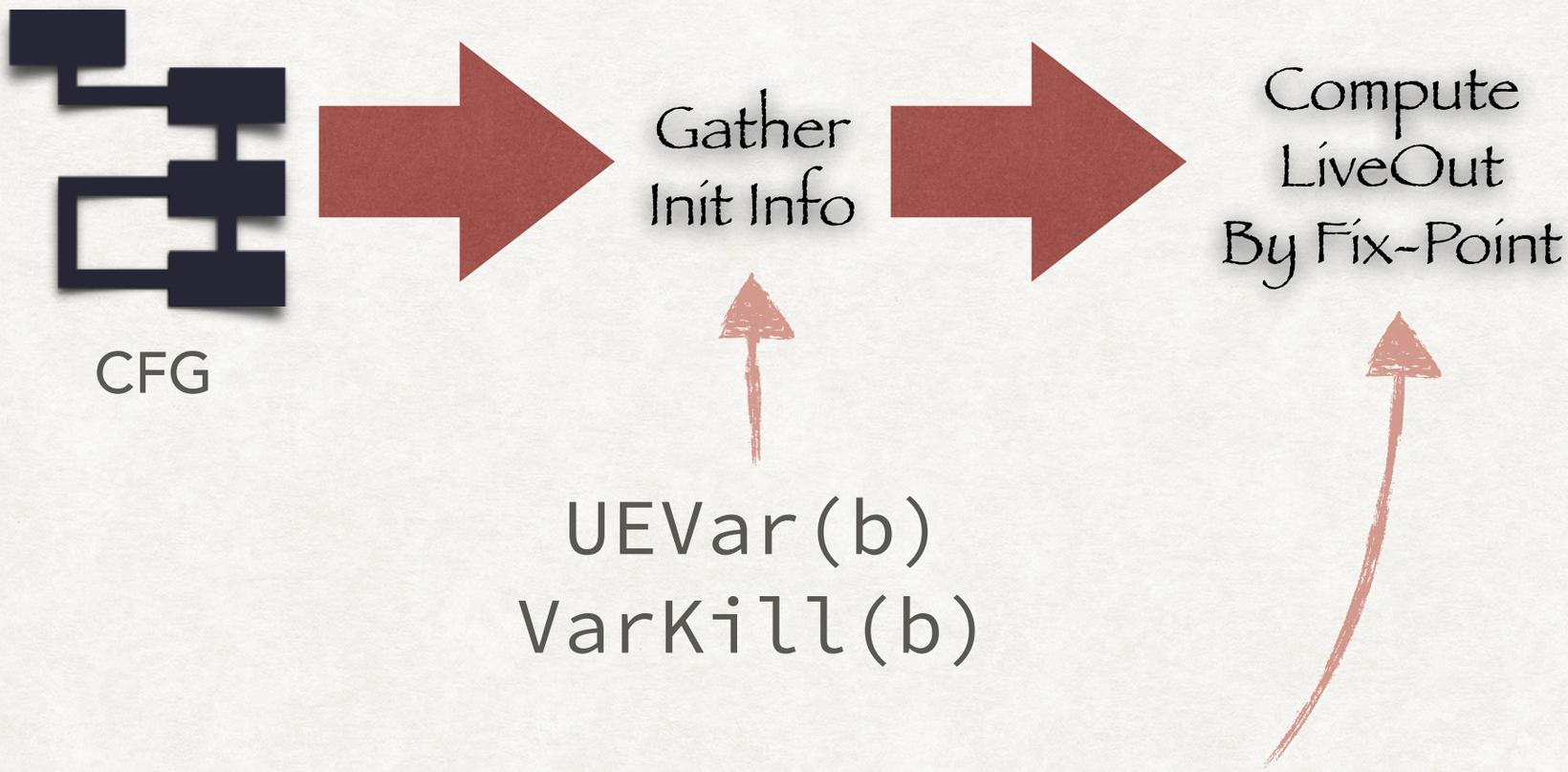
we want to know how long a variable will live to allocate regs



Liveness Analysis

Why do we want to do Liveness Analysis?

we want to know how long a variable will live to allocate regs

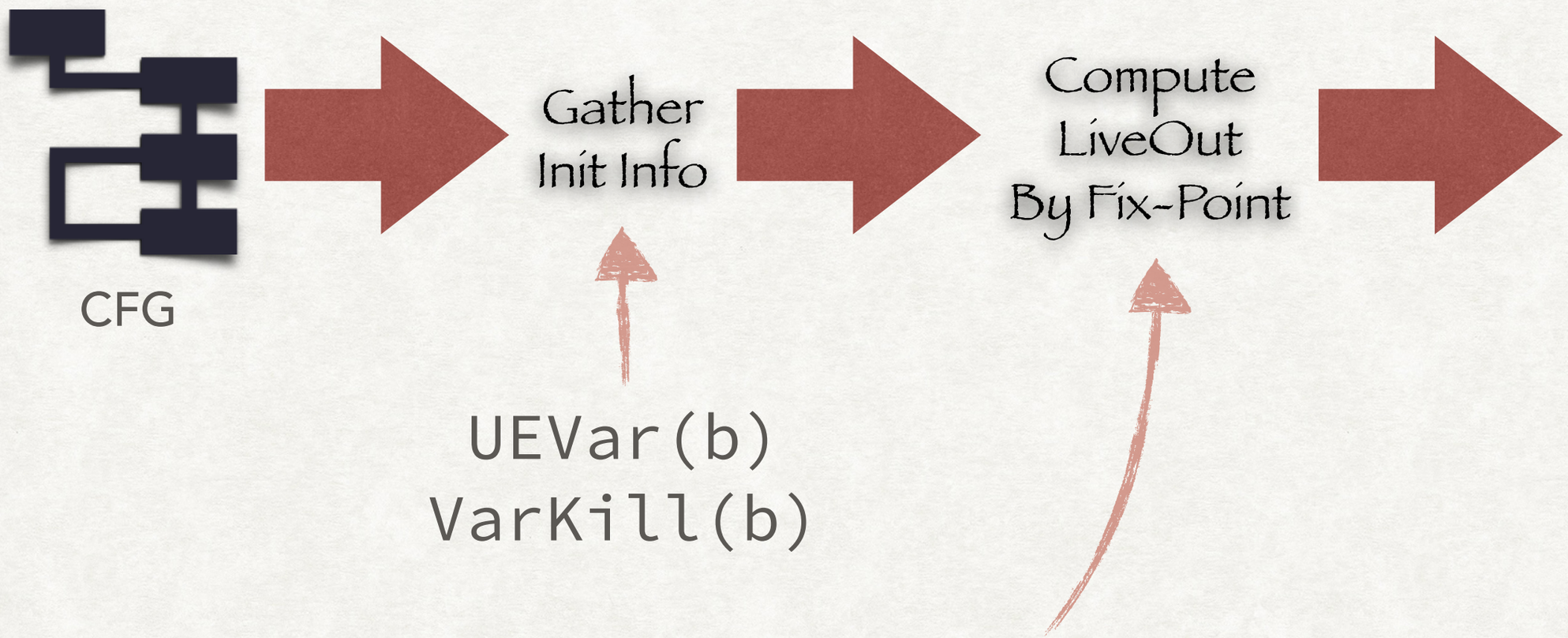


$$\mathtt{LiveOut(b)} = \bigcup_{m \in succ(b)} (\mathtt{UEVar(m)} \cup (\mathtt{LiveOut(m)} \cap \mathtt{VarKill(m)}^c))$$

Liveness Analysis

Why do we want to do Liveness Analysis?

we want to know how long a variable will live to allocate regs

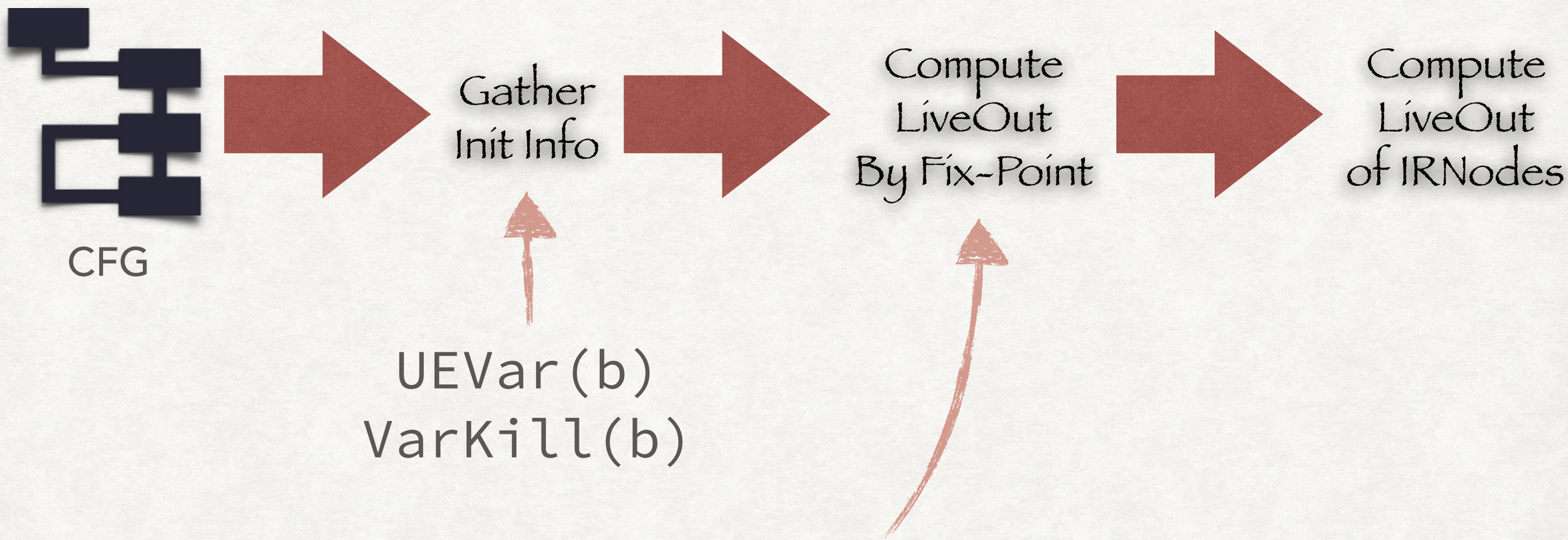


$$\mathtt{LiveOut(b)} = \bigcup_{m \in succ(b)} (\mathtt{UEVar(m)} \cup (\mathtt{LiveOut(m)} \cap \mathtt{VarKill(m)}^c))$$

Liveness Analysis

Why do we want to do Liveness Analysis?

we want to know how long a variable will live to allocate regs

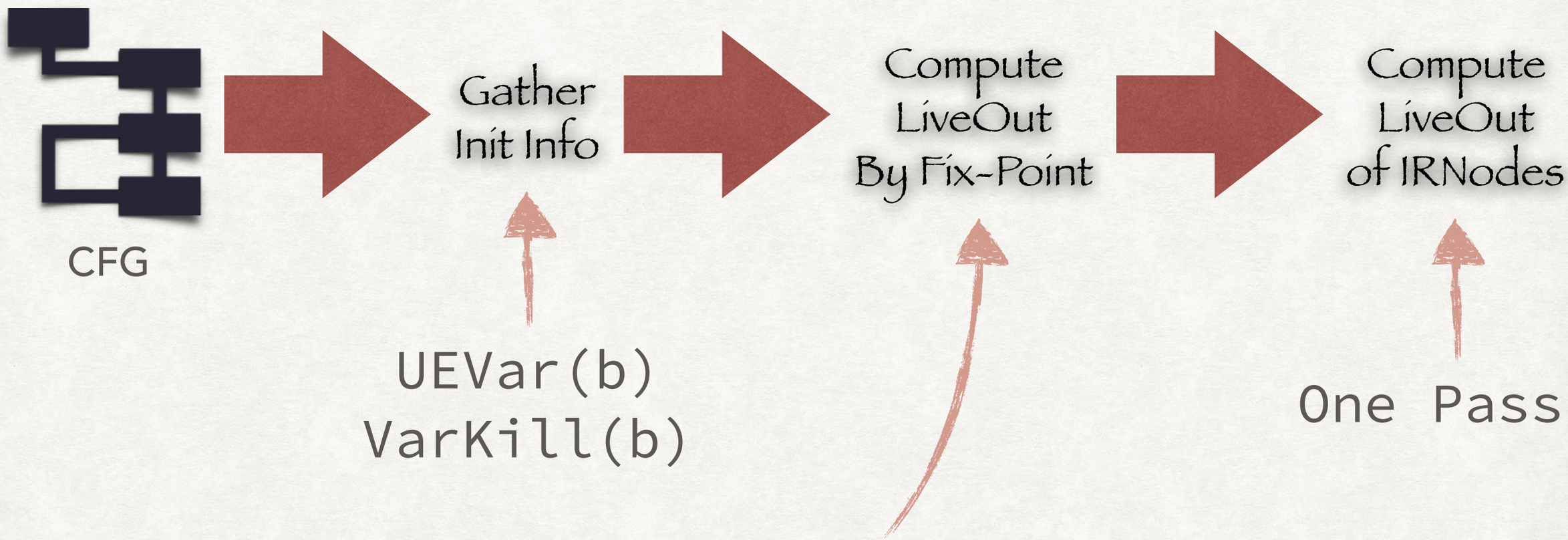


$$\mathrm{LiveOut}(b) = \bigcup_{m \in \mathrm{succ}(b)} (\mathrm{UEVar}(m) \cup (\mathrm{LiveOut}(m) \cap \mathrm{VarKill}(m)^c))$$

Liveness Analysis

Why do we want to do Liveness Analysis?

we want to know how long a variable will live to allocate regs



$$\text{LiveOut}(b) = \bigcup_{m \in succ(b)} (\text{UEVar}(m) \cup (\text{LiveOut}(m) \cap \text{VarKill}(m)^c))$$

Register Allocation

Register Allocation

Coloring:

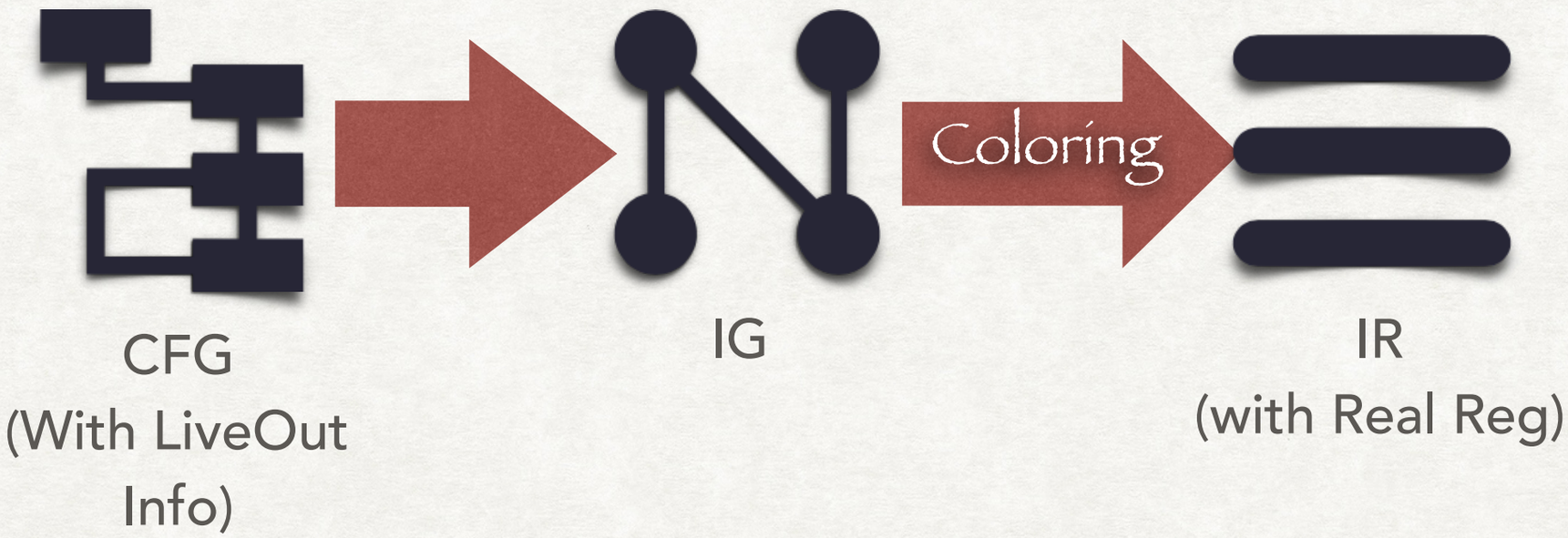
Register Allocation

Coloring:

to allocate not-limited regs into limited amount of regs

Register Allocation

Coloring:
to allocate not-limited regs into limited amount of regs



Register Allocation

Register Allocation

Strategy:

Register Allocation

Strategy:

1. put all regs into a priorityQueue, sort them by their degrees in IG in ascending order

Register Allocation

Strategy:

1. put all regs into a priorityQueue, sort them by their degrees in IG in ascending order
2. poll regs one by one from priorityQueue and put them into a stack. At the same time, delete the reg in the IG, update the priorityQueue

Register Allocation

Strategy:

1. put all regs into a priorityQueue, sort them by their degrees in IG in ascending order
2. poll regs one by one from priorityQueue and put them into a stack. At the same time, delete the reg in the IG, update the priorityQueue
3. pop regs from the stack one by one, color them. If spill, tag it.

Register Allocation

Strategy:

1. put all regs into a priorityQueue, sort them by their degrees in IG in ascending order
2. poll regs one by one from priorityQueue and put them into a stack. At the same time, delete the reg in the IG, update the priorityQueue
3. pop regs from the stack one by one, color them. If spill, tag it.

I don't need to worry about spill, for I reserve 2 regs for load/store!

Optimization — 3 Little things

Optimization — 3 Little things

Optimizations I have done:

Optimization — 3 Little things

Optimizations I have done:

1. Dead Code Elimination

Optimization — 3 Little things

Optimizations I have done:

1. Dead Code Elimination

```
if (node.def ∉ node.LiveOut)
```

Optimization — 3 Little things

Optimizations I have done:

1. Dead Code Elimination

```
if (node.def ∉ node.LiveOut)
    delete node
```

Optimization — 3 Little things

Optimizations I have done:

1. Dead Code Elimination

```
if (node.def ∉ node.LiveOut)
    delete node
```

2. Branch Expressing Refine

Optimization — 3 Little things

Optimizations I have done:

1. Dead Code Elimination

```
if (node.def ∉ node.LiveOut)
    delete node
```

2. Branch Expressing Refine

   if there is fallThrough opportunity, eliminate a `Jump`

Optimization — 3 Little things

Optimizations I have done:

1. Dead Code Elimination

```
if (node.def ∉ node.LiveOut)
    delete node
```

2. Branch Expressing Refine

   if there is fallThrough opportunity, eliminate a `Jump`

3. String Optimization

Optimization — 3 Little things

Optimizations I have done:

1. Dead Code Elimination

```
if (node.def ∉ node.LiveOut)
    delete node
```

2. Branch Expressing Refine

   if there is fallThrough opportunity, eliminate a `Jump`

3. String Optimization

```
println("four"
+ " " +
"score")
```

Optimization — 3 Little things

Optimizations I have done:

1. Dead Code Elimination

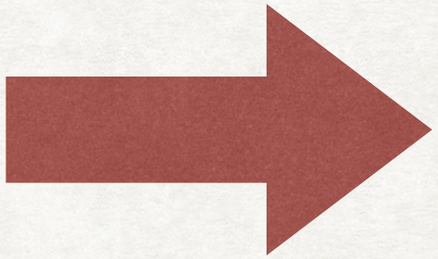
```
if (node.def ∉ node.LiveOut)
    delete node
```

2. Branch Expressing Refine

   if there is fallThrough opportunity, eliminate a `Jump`

3. String Optimization

```
println("four"
+ " " +
"score")
```



Optimization — 3 Little things

Optimizations I have done:

1. Dead Code Elimination

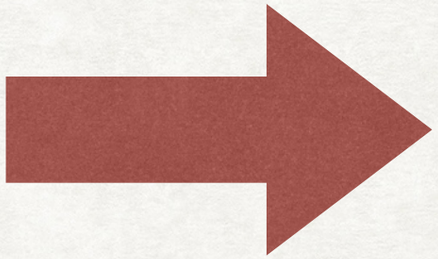
```
if (node.def ∉ node.LiveOut)
    delete node
```

2. Branch Expressing Refine

   if there is fallThrough opportunity, eliminate a `Jump`

3. String Optimization

```
println("four"
+ " " +
"score")
```



```
print("four");
print(" ");
print("score");
println("");
```

Optimization

Optimization

Why optimize strings like this?

Optimization

Why optimize strings like this?

1. builtIn functions like `stringConcatenate` and `toString` are expensive!

Optimization

Why optimize strings like this?

1. builtIn functions like `stringConcatenate` and `toString` are expensive !
2. planned to implement sth. like `StringBuilder` but do not have enough time…

Optimization

Why optimize strings like this?

1. builtIn functions like `stringConcatenate` and `toString` are expensive!
2. planned to implement sth. like `StringBuilder` but do not have enough time…

Java:

Optimization

Why optimize strings like this?

1. builtIn functions like `stringConcatenate` and `toString` are expensive !
2. planned to implement sth. like `StringBuilder` but do not have enough time…

Java:

```
“four” + “ ” + “score”
```

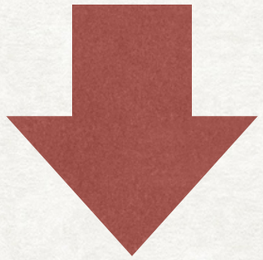
Optimization

Why optimize strings like this?

1. builtIn functions like `stringConcatenate` and `toString` are expensive !
2. planned to implement sth. like `StringBuilder` but do not have enough time…

Java:

```
"four" + " " + "score"
```



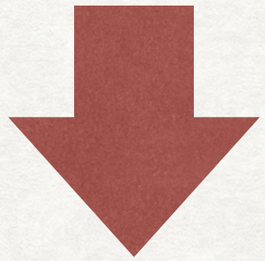
Optimization

Why optimize strings like this?

1. builtIn functions like `stringConcatenate` and `toString` are expensive !
2. planned to implement sth. like `StringBuilder` but do not have enough time…

Java:

```
"four" + " " + "score"
```



```
StringBuilder x;
X.append("four").append(" ").append("score");
```

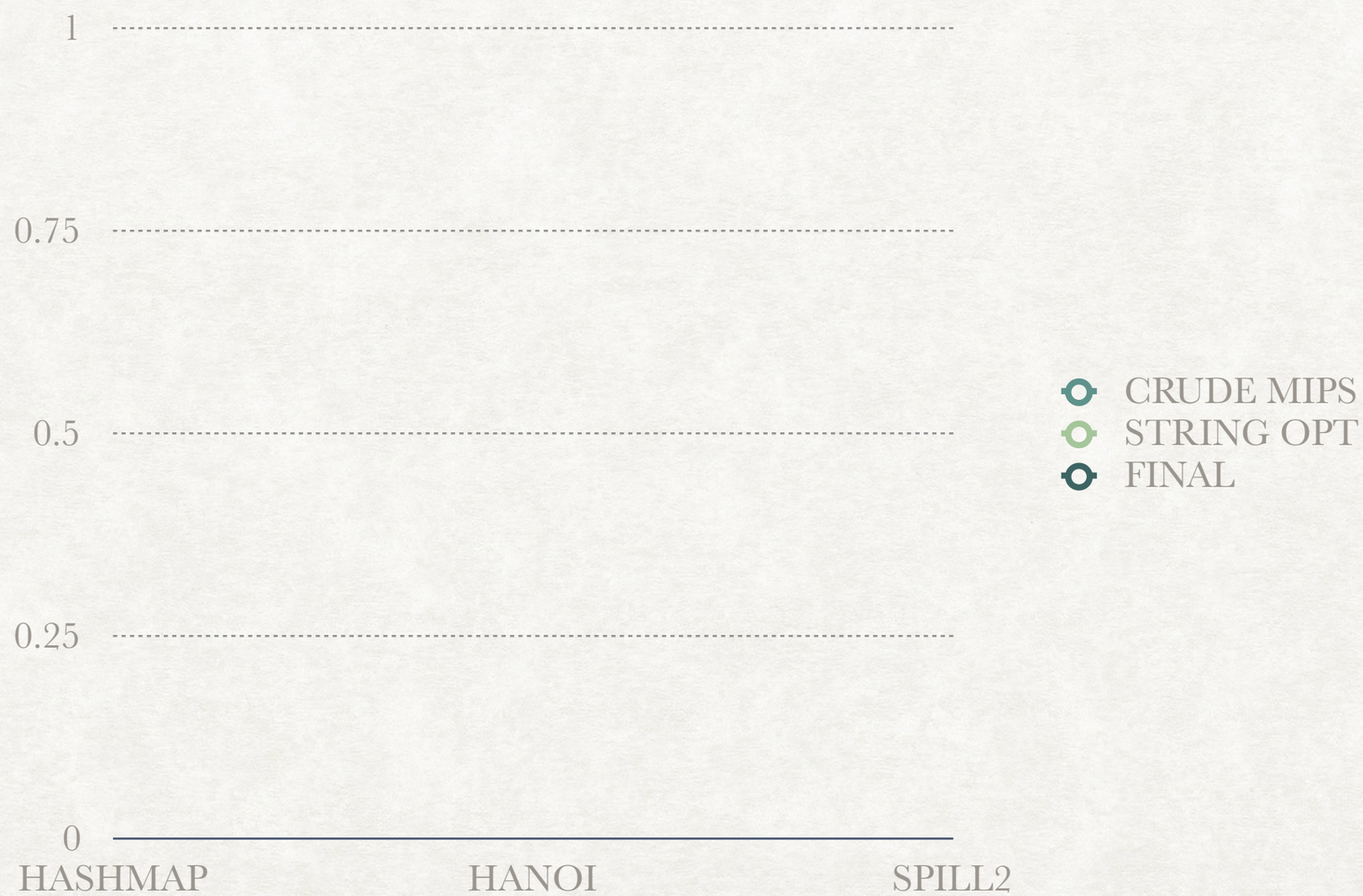
Optimization

Optimization

Performance Improvement:

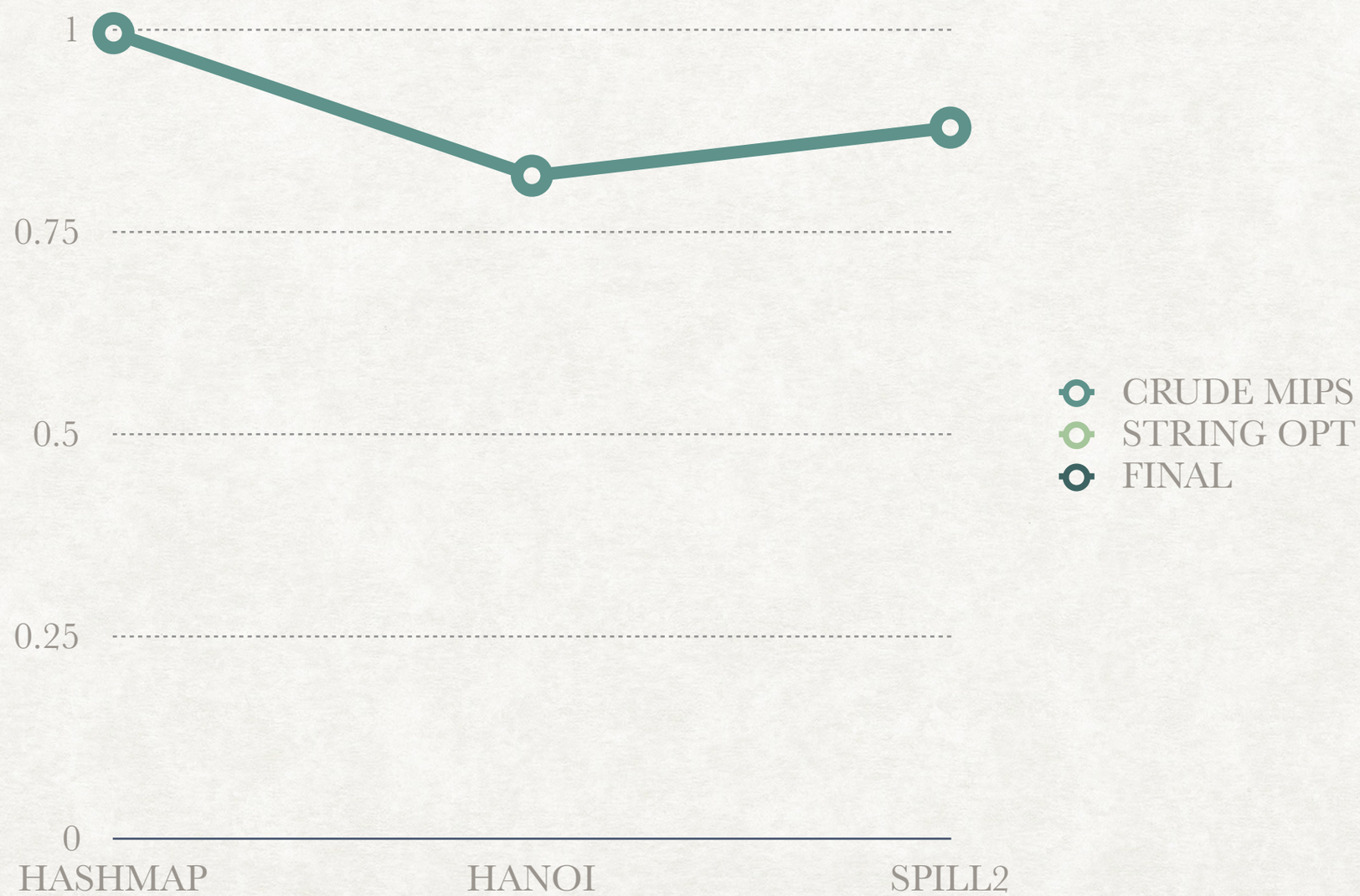
Optimization

Performance Improvement:



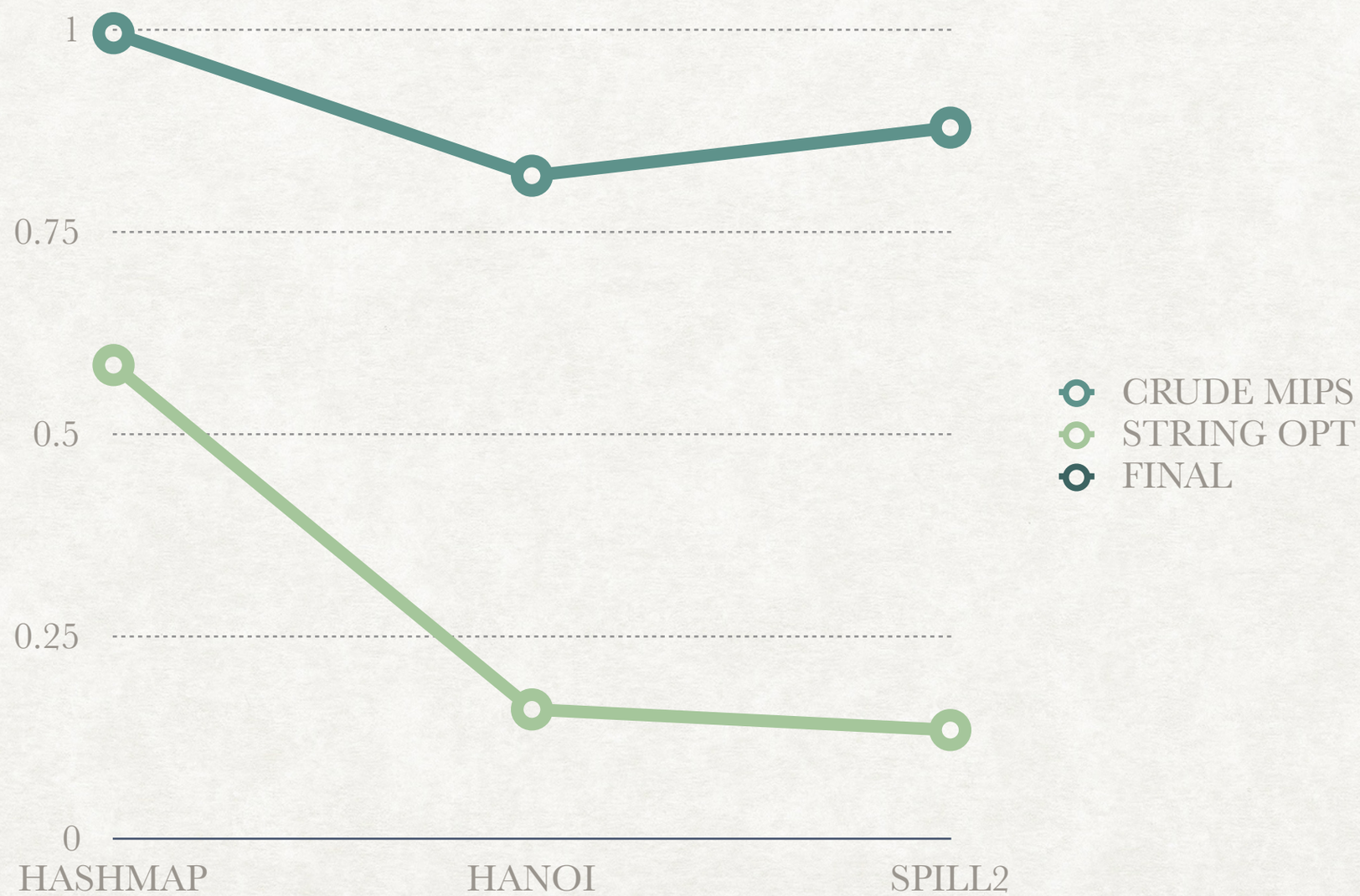
Optimization

Performance Improvement:



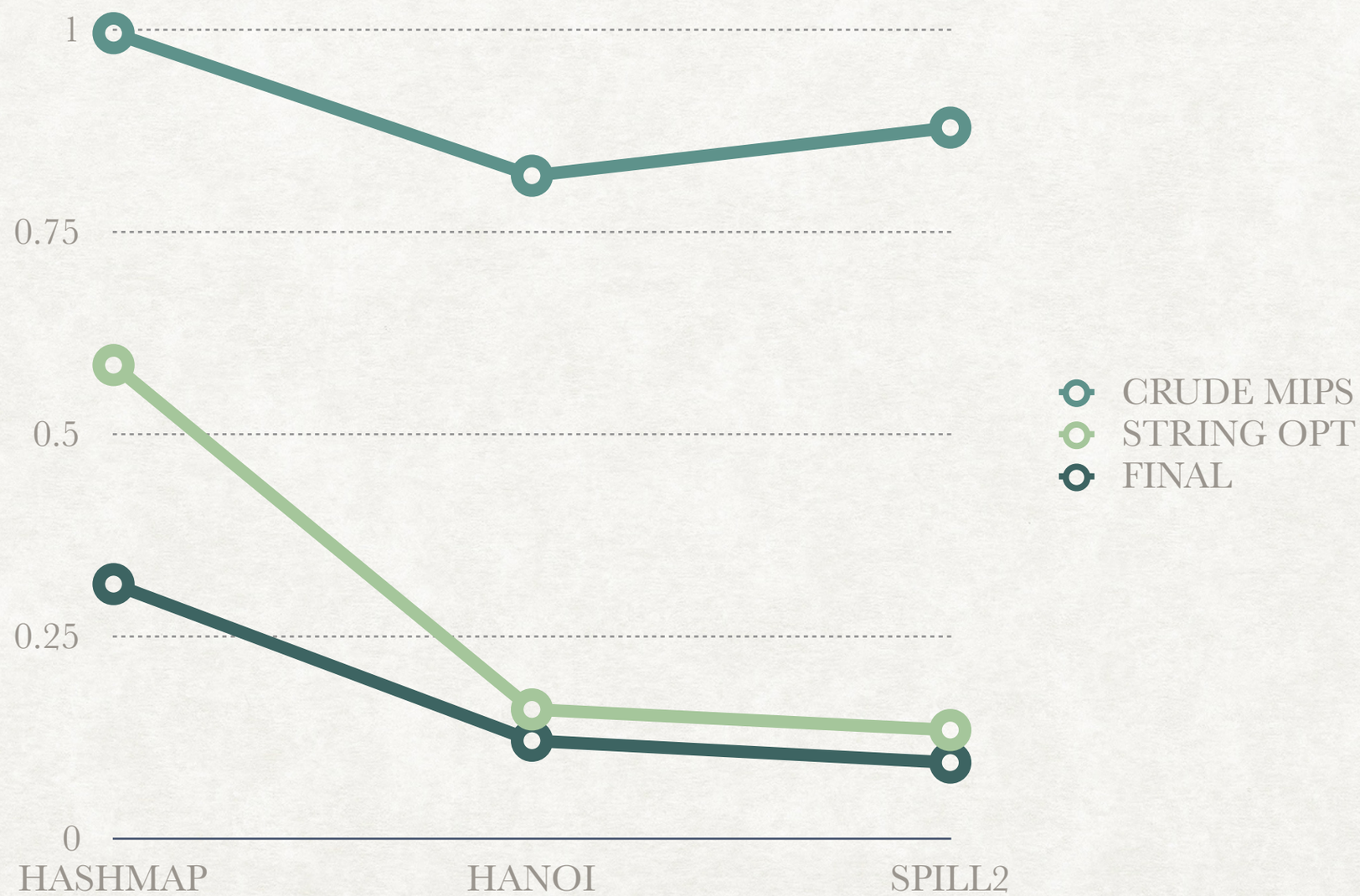
Optimization

Performance Improvement:



Optimization

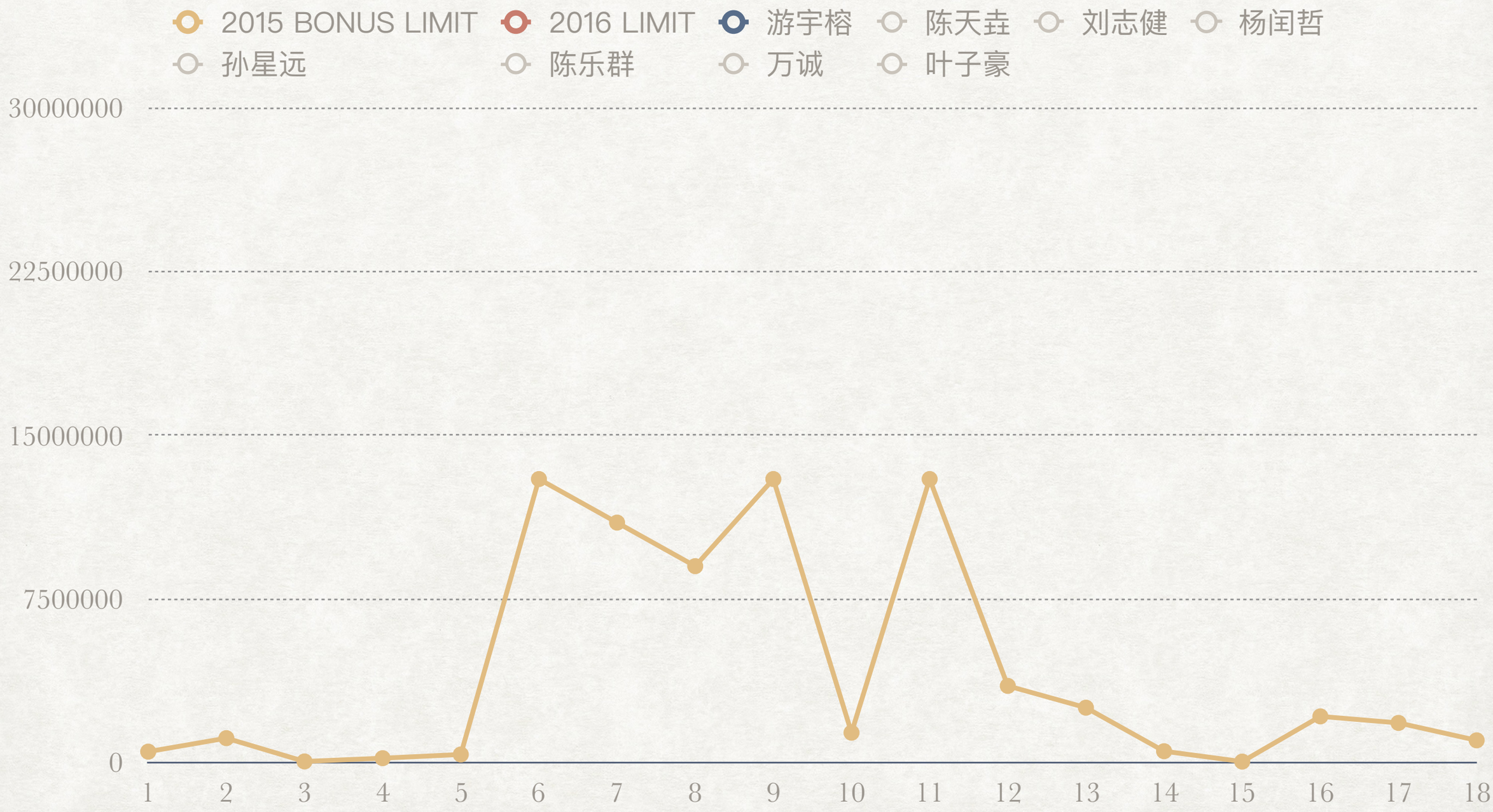
Performance Improvement:



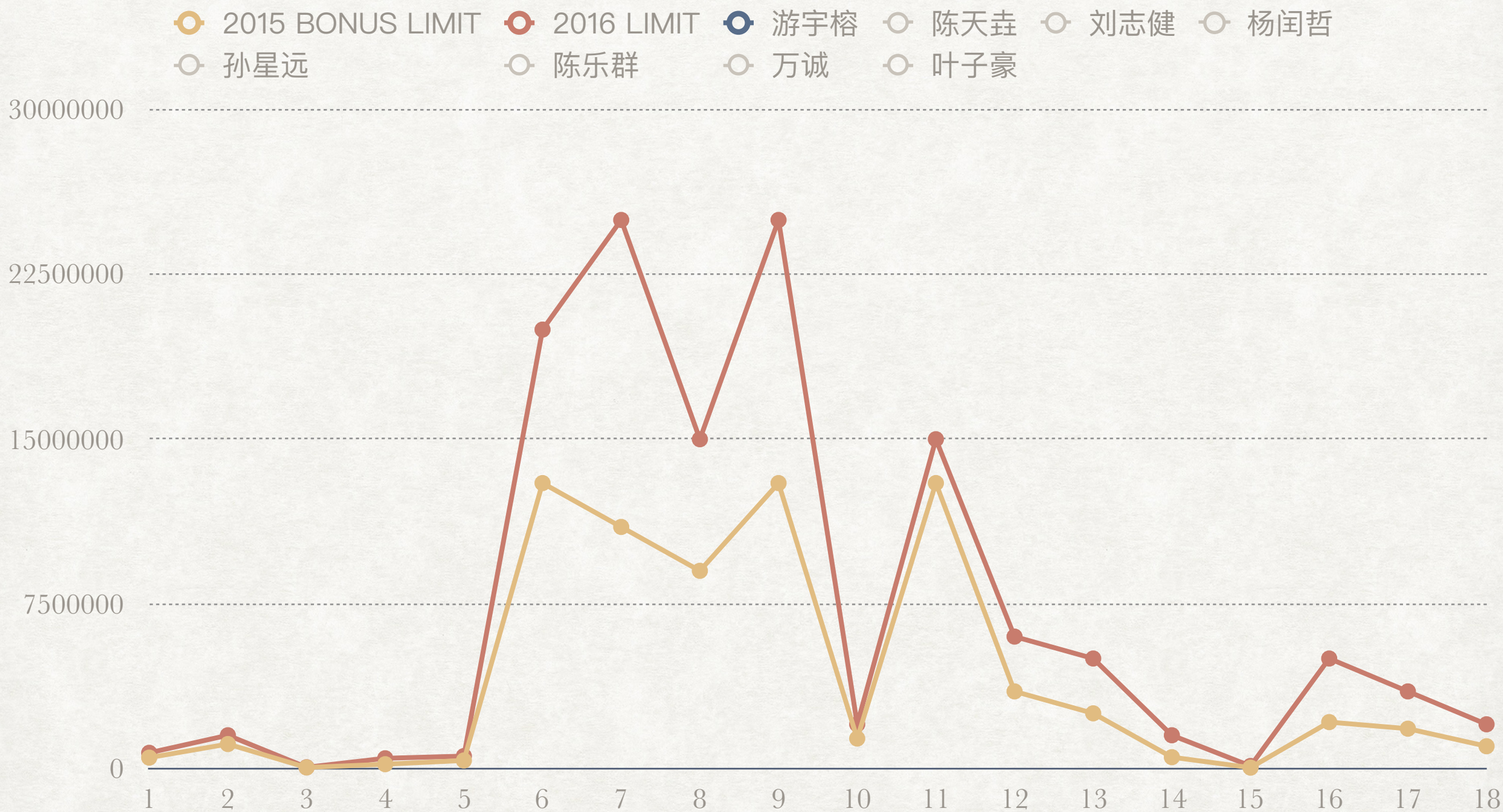
Overall Performance

Overall Performance

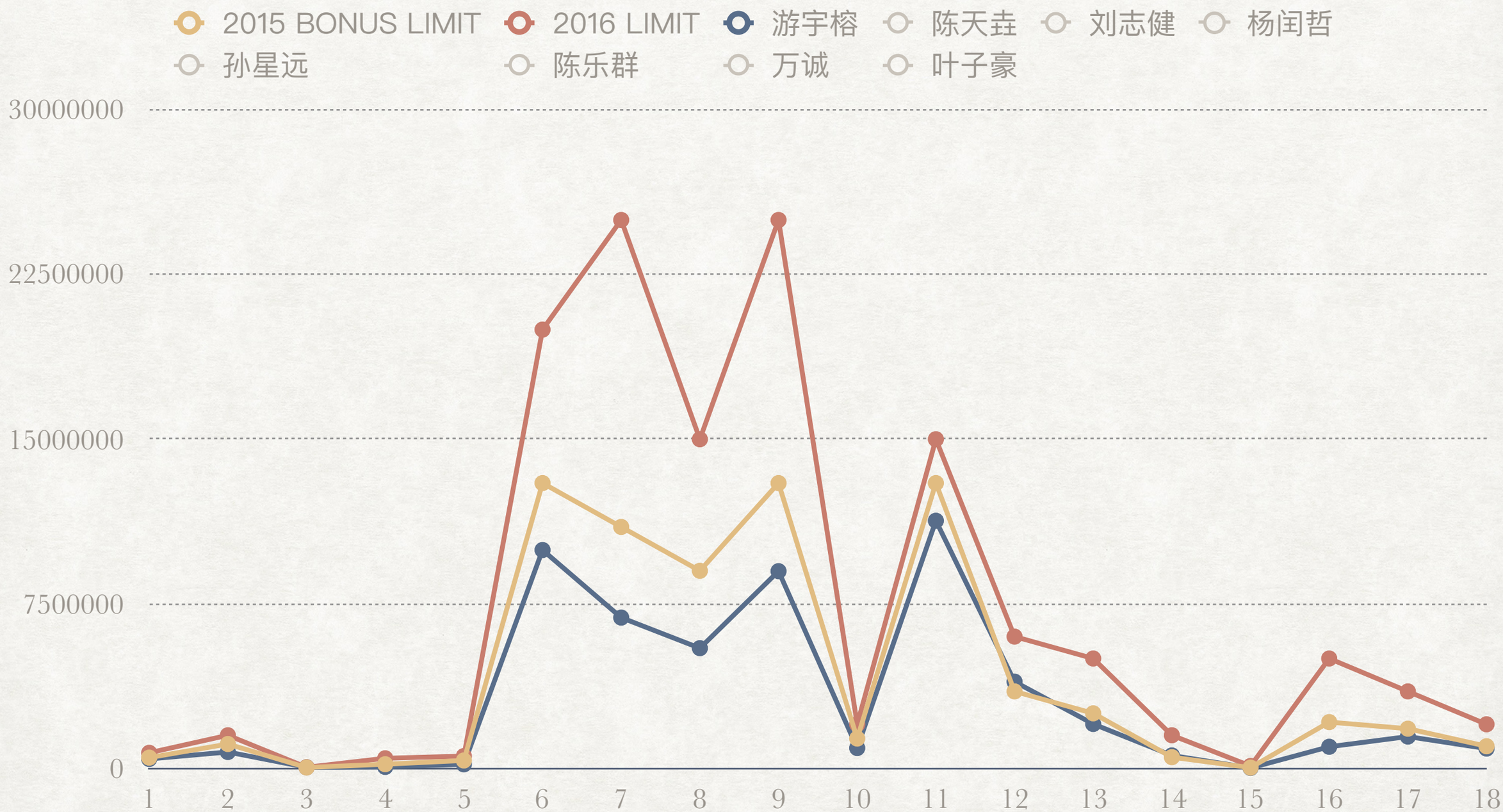
Overall Performance



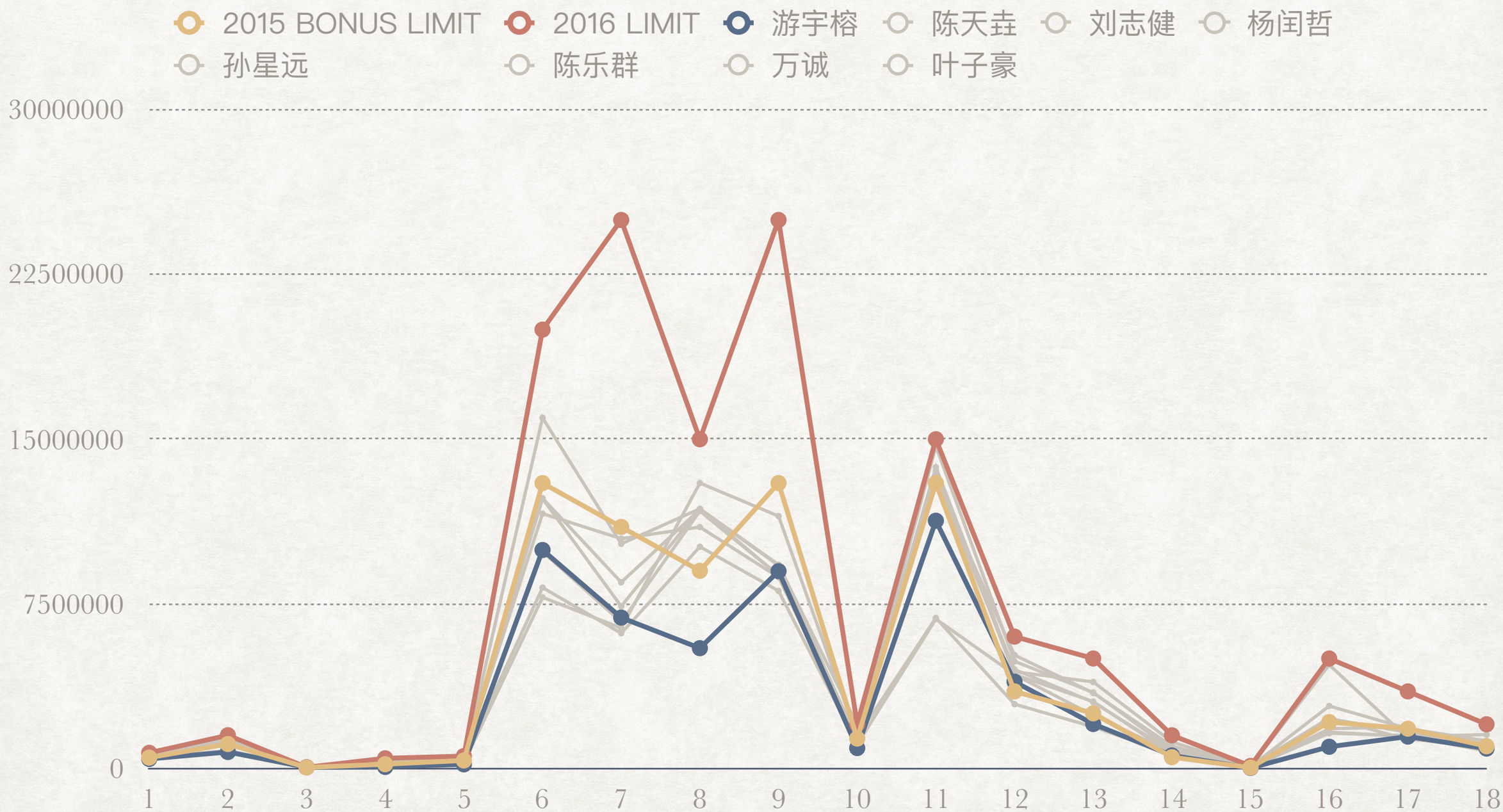
Overall Performance



Overall Performance



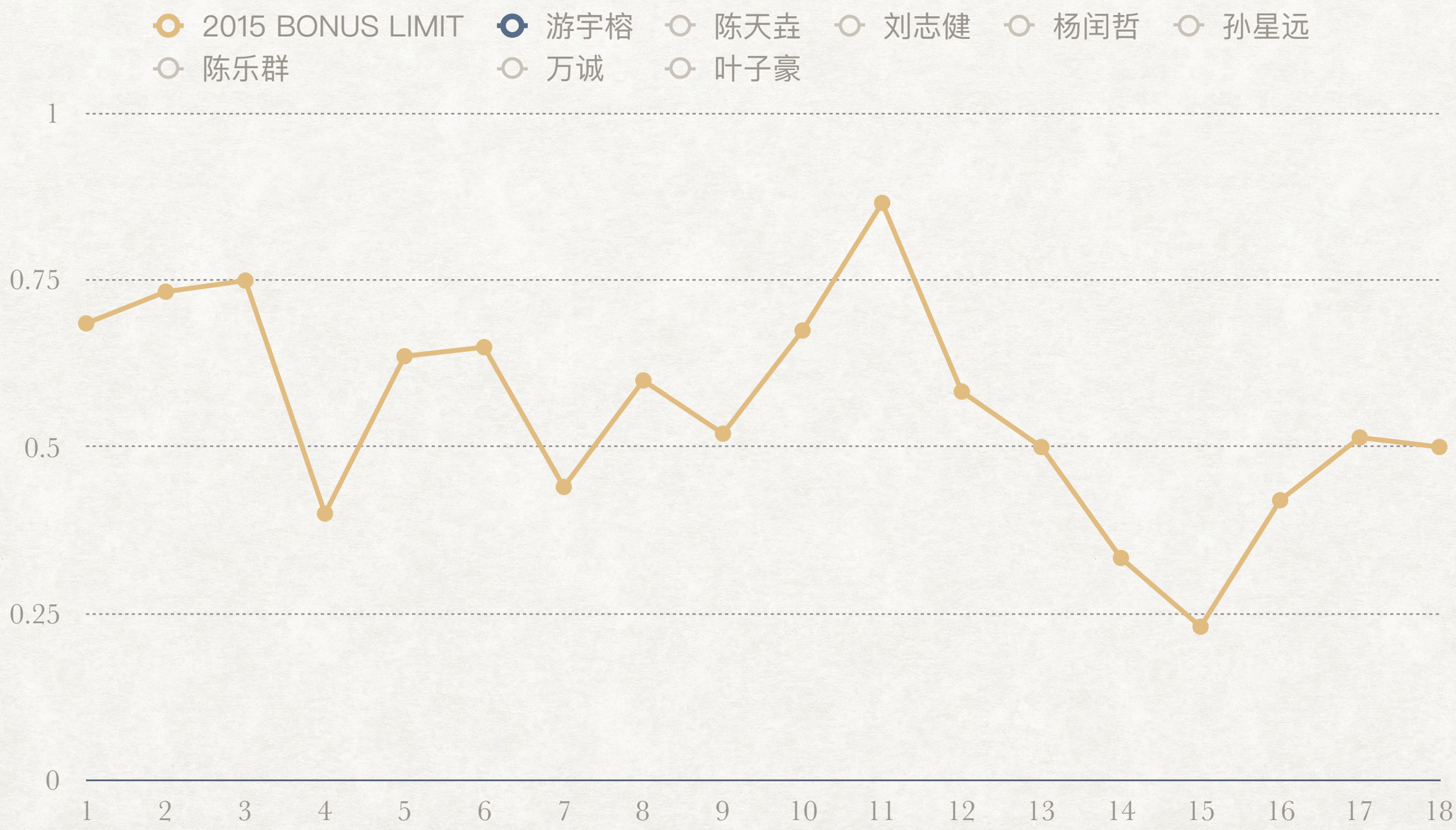
Overall Performance



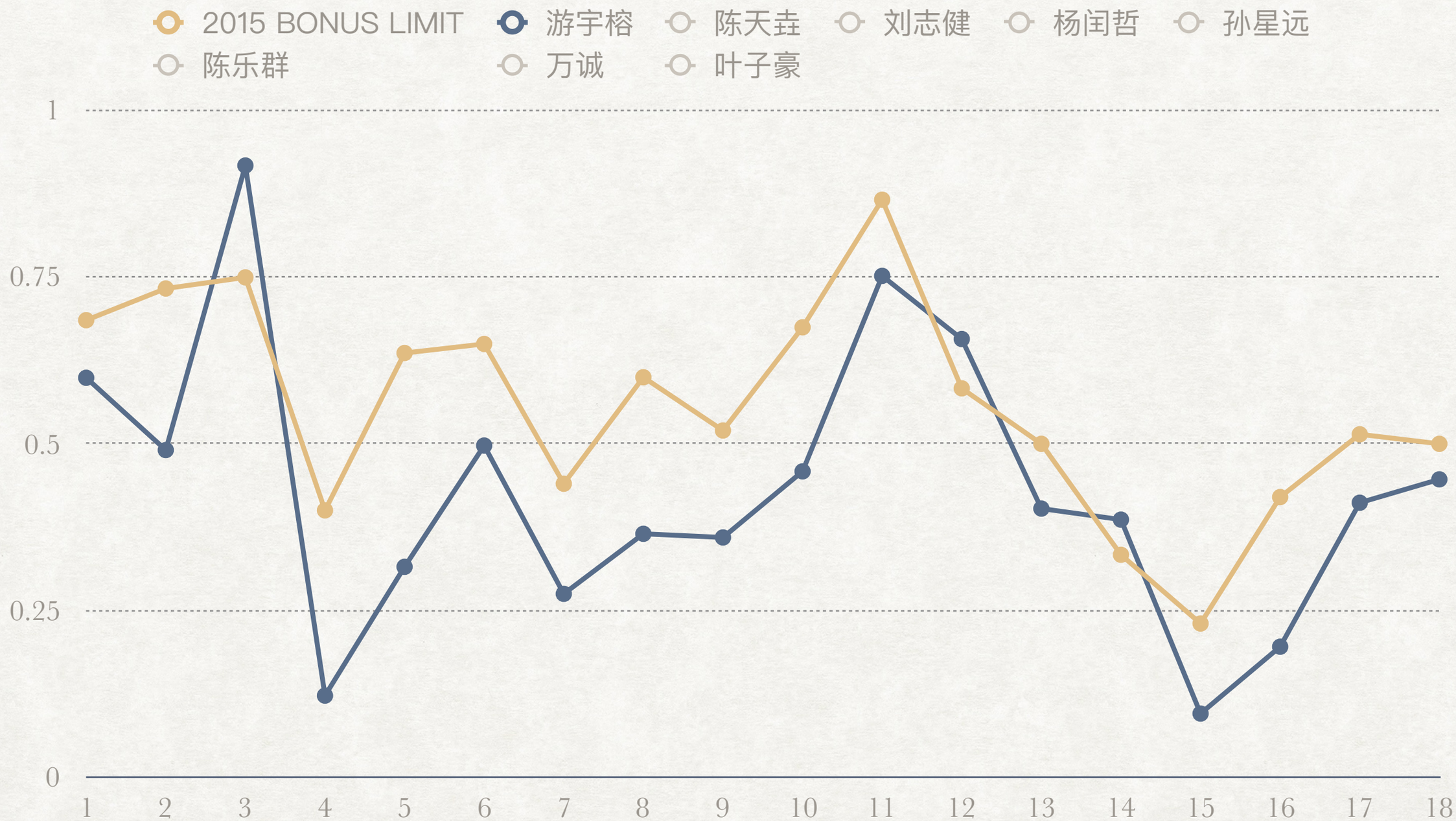
Overall Performance

Overall Performance

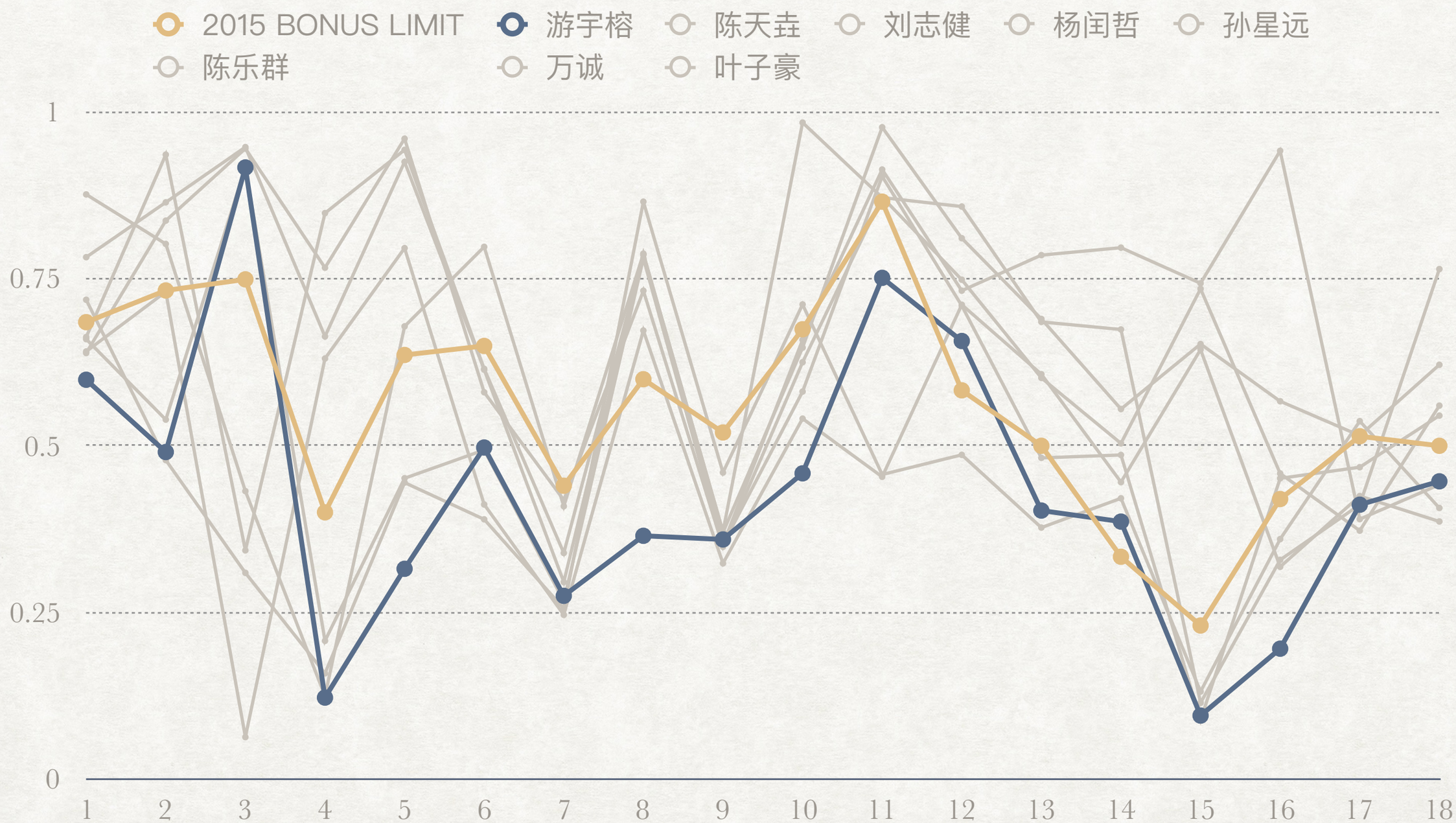
Overall Performance



Overall Performance



Overall Performance



An Attempt to OO

An Attempt to OO

Support for Class Function:

An Attempt to OO

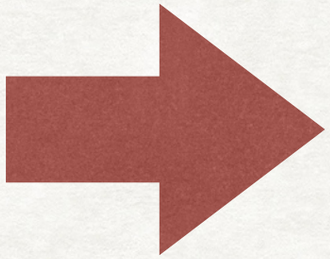
Support for Class Function:

```
class A{
  int a;
  int f(){
    return a + 1;
  }
}
```

An Attempt to OO

Support for Class Function:

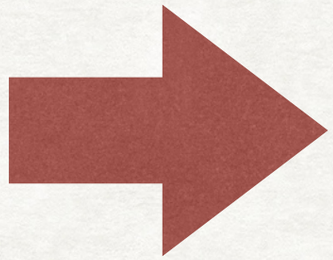
```
class A{
  int a;
  int f(){
    return a + 1;
  }
}
```



An Attempt to OO

Support for Class Function:

```
class A{
  int a;
  int f(){
    return a + 1;
  }
}
```

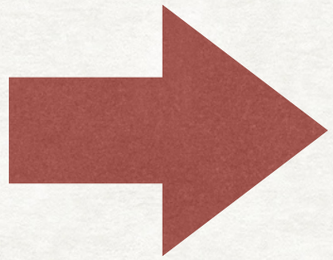


```
int A.f(A this){
  return this.a + 1;
}
```

An Attempt to OO

Support for Class Function:

```
class A{
  int a;
  int f(){
    return a + 1;
  }
}
```



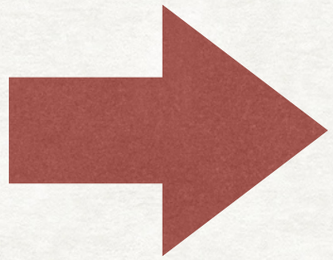
```
int A.f(A this){
  return this.a + 1;
}
```

All transformation done in AST phrase.

An Attempt to OO

Support for Class Function:

```
class A{
  int a;
  int f(){
    return a + 1;
  }
}
```



```
int A.f(A this){
  return this.a + 1;
}
```

All transformation done in AST phrase.
Class function have no more difference between ordinary function.

Imperfections

Imperfections

Some regrets due to the lack of developing time.

Imperfections

Some regrets due to the lack of developing time.

1. Simpler design on SymbolTable: AST node as Key

Imperfections

Some regrets due to the lack of developing time.

1. Simpler design on SymbolTable: AST node as Key
2. SSA form

Imperfections

Some regrets due to the lack of developing time.

1. Simpler design on SymbolTable: AST node as Key
2. SSA form
3. decouple the phrases

Imperfections

Some regrets due to the lack of developing time.

1. Simpler design on SymbolTable: AST node as Key
2. SSA form
3. decouple the phrases

   for example, optimizations should be done solely on a visitor on AST/IR, but in order to develop quickly, I directly perform them on AST/IR builder

Imperfections

Some regrets due to the lack of developing time.

1. Simpler design on SymbolTable: AST node as Key
2. SSA form
3. decouple the phrases
   for example, optimizations should be done solely on a visitor on AST/IR, but in order to develop quickly, I directly perform them on AST/IR builder
4. A more careful coloring strategy

Imperfections

Some regrets due to the lack of developing time.

1. Simpler design on SymbolTable: AST node as Key
2. SSA form
3. decouple the phrases
   for example, optimizations should be done solely on a visitor on AST/IR, but in order to develop quickly, I directly perform them on AST/IR builder
4. A more careful coloring strategy
5. StringBuilder

Imperfections

Some regrets due to the lack of developing time.

1. Simpler design on SymbolTable: AST node as Key
2. SSA form
3. decouple the phrases
   for example, optimizations should be done solely on a visitor on AST/IR, but in order to develop quickly, I directly perform them on AST/IR builder
4. A more careful coloring strategy
5. StringBuilder
6. Inherence, Polymorphism

Imperfections

Some regrets due to the lack of developing time.

1. Simpler design on SymbolTable: AST node as Key
2. SSA form
3. decouple the phrases
   for example, optimizations should be done solely on a visitor on AST/IR, but in order to develop quickly, I directly perform them on AST/IR builder
4. A more careful coloring strategy
5. StringBuilder
6. Inherence, Polymorphism
7. Garbage Collector

What I have learned:

What I have learned:

1. Much more confident in coding

What I have learned:

1. Much more confident in coding
2. Basic impression on how to develop a big project

What I have learned:

1. Much more confident in coding
2. Basic impression on how to develop a big project
3. Java — it's really a modern language!

What I have learned:

1. Much more confident in coding
2. Basic impression on how to develop a big project
3. Java — it's really a modern language!
4. Design patterns: visitor, listener, ...

What I have learned:

1. Much more confident in coding
2. Basic impression on how to develop a big project
3. Java — it's really a modern language!
4. Design patterns: visitor, listener, ...
5. Classic problems' modern applications:

What I have learned:

1. Much more confident in coding
2. Basic impression on how to develop a big project
3. Java — it's really a modern language!
4. Design patterns: visitor, listener, ……
5. Classic problems' modern applications:
   DFA, CFG, Graph Coloring ……

What I have learned:

1. Much more confident in coding
2. Basic impression on how to develop a big project
3. Java — it's really a modern language!
4. Design patterns: visitor, listener, ...
5. Classic problems' modern applications:
   DFA, CFG, Graph Coloring .....
6. Details hidden in programing language

What I have learned:

1. Much more confident in coding
2. Basic impression on how to develop a big project
3. Java — it's really a modern language!
4. Design patterns: visitor, listener, ...
5. Classic problems' modern applications:
   DFA, CFG, Graph Coloring .....
6. Details hidden in programing language
   How complex emerge from simplicity; Abstraction;

What I have learned:

1. Much more confident in coding
2. Basic impression on how to develop a big project
3. Java — it's really a modern language!
4. Design patterns: visitor, listener, ……
5. Classic problems' modern applications:
   DFA, CFG, Graph Coloring ……
6. Details hidden in programing language
   How complex emerge from simplicity; Abstraction;
   mechanisms of class, array, functions, iterations……

What I have learned:

1. Much more confident in coding
2. Basic impression on how to develop a big project
3. Java — it's really a modern language!
4. Design patterns: visitor, listener, ...
5. Classic problems' modern applications:
   DFA, CFG, Graph Coloring .....
6. Details hidden in programing language
   How complex emerge from simplicity; Abstraction;
   mechanisms of class, array, functions, iterations.....
7. Write assembly language for the first time

Reflections

Reflections

1. Why did this projects move on so smoothly?

Reflections

1. Why did this projects move on so smoothly?
    - Head Start: began to learn Java since winter vacation

Reflections

1. Why did this projects move on so smoothly?
   - Head Start: began to learn Java since winter vacation
   - Frequent discusses between classmates, TAs & Boss Ma

Reflections

1. Why did this projects move on so smoothly?
   - Head Start: began to learn Java since winter vacation
   - Frequent discusses between classmates, TAs & Boss Ma
   - Special thanks to 叶子豪，陈天垚，刘志健

Reflections

1. Why did this projects move on so smoothly?
   - Head Start: began to learn Java since winter vacation
   - Frequent discusses between classmates, TAs & Boss Ma
     - Special thanks to 叶子豪，陈天垚，刘志健 and 马豪君

Reflections

1. Why did this projects move on so smoothly?
   - Head Start: began to learn Java since winter vacation
   - Frequent discusses between classmates, TAs & Boss Ma
     - Special thanks to 叶子豪，陈天垚，刘志健 and 马豪君
   - Massive Reading: Tiger Book, EaC, MIPS tutorials…

Reflections

1. Why did this projects move on so smoothly?
    - Head Start: began to learn Java since winter vacation
    - Frequent discusses between classmates, TAs & Boss Ma
        - Special thanks to 叶子豪，陈天垚，刘志健 and 马豪君
    - Massive Reading: Tiger Book, EaC, MIPS tutorials…
2. Thinking clearly, deep and far is of the most importance

Reflections

1. Why did this projects move on so smoothly?
   - Head Start: began to learn Java since winter vacation
   - Frequent discusses between classmates, TAs & Boss Ma
     - Special thanks to 叶子豪，陈天垚，刘志健 and 马豪君
   - Massive Reading: Tiger Book, EaC, MIPS tutorials…
2. Thinking clearly, deep and far is of the most importance
3. Good designs can only be created by far-sighted designer.

Reflectio

on

s

2.

3.

CHEERS & THANKS!