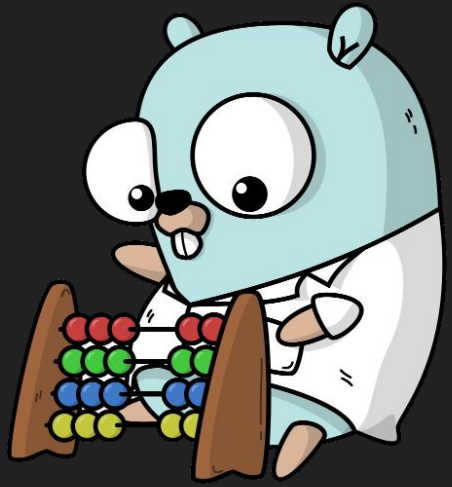
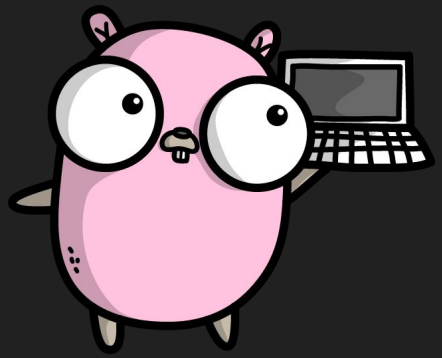
Optimizing Performance

using a VM and Go Plugins



Travis Smith
Senior Software Architect at Workiva
GopherCon 2020

Our Agenda



A1 : Spreadsheet basics

A2 : How to calculate formulas

A3 : Design and build a virtual machine

A4 : Compiling Go Plugins at runtime

A5 : SUM(A1:A4) > VALUE()

Spreadsheet Basics

Spreadsheet Basics

- A spreadsheet arranges data in a table of cells

Spreadsheet Basics

- A spreadsheet arranges data in a table of cells

Assets		
	12/31/2017	12/31/2016
Current Assets		
Cash	60,000	100,000
Petty Cash	5,000	1,000
Accounts Receivable - ne	40,000	65,000
Inventory	10,000	12,000
Supplies	3,000	2,500
Prepaid Insurance	12,000	4,500
Total current assets	130,000	185,000
Property, plant and equipment		
Land	30,000	30,000
Land improvements	4,500	4,000
Buildings	65,000	65,000
Equipment	26,000	8,000
Less: accumulated deprec	(25,000)	(20,000)
Prop, plant, and equip-ne	100,500	87,000

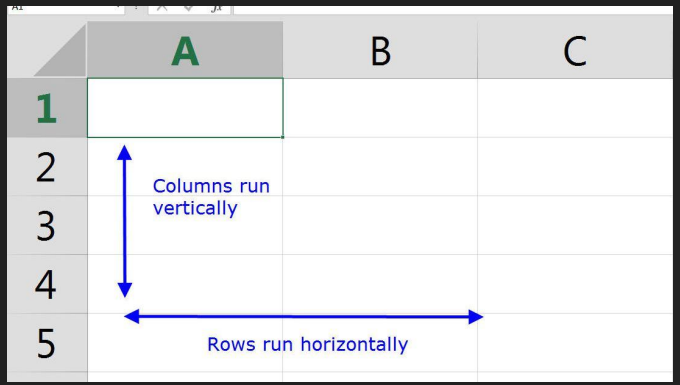
Liabilities		
	12/31/2017	12/31/2016
Current Liabilities		
Notes payable	2,500	4,000
Accounts payable	50,000	80,000
Wages payable	35,000	42,000
Interest payable	3,000	3,200
Taxes payable	500	1,000
Unearned revenues	1,500	1,300
Total current liabilities	92,500	131,500
Long-term liabilities		
Notes payable	12,000	22,000
Bonds payable	75,000	100,000
Total long-term liabilities	87,000	122,000
Total liabilities	179,500	253,500

Spreadsheet Basics

- A spreadsheet arranges data in a table of cells
- Cells are arranged in a grid of rows and columns

Spreadsheet Basics

- A spreadsheet arranges data in a table of cells
- Cells are arranged in a grid of rows and columns

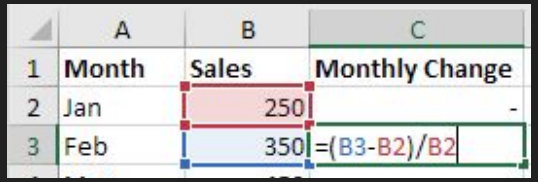


Spreadsheet Basics

- A spreadsheet arranges data in a table format
- Cells are arranged in a grid of rows and columns
- Every cell can hold either a value or a formula

Spreadsheet Basics

- A spreadsheet arranges data in a table format
- Cells are arranged in a grid of rows and columns
- Every cell can hold either a value or a formula



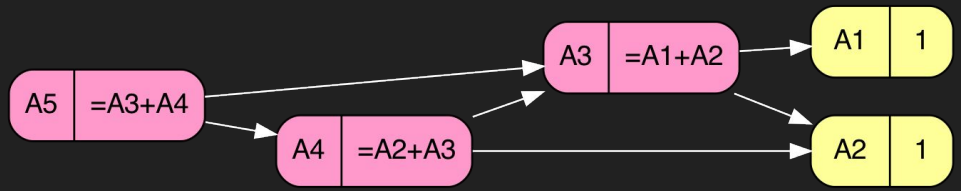
	A	B	C
1	Month	Sales	Monthly Change
2	Jan	250	-
3	Feb	350	=(B3-B2)/B2

Spreadsheet Basics

- A spreadsheet arranges data in a table format
- Cells are arranged in a grid of rows and columns
- Every cell can hold either a value or a formula
- Formulas can chain together by referencing other cells

Spreadsheet Basics

- A spreadsheet arranges data in a table format
- Cells are arranged in a grid of rows and columns
- Every cell can hold either a value or a formula
- Formulas can chain together by referencing other cells



Fibonacci Sequence

The ability to

chain 🔗 formulas 🔗 together

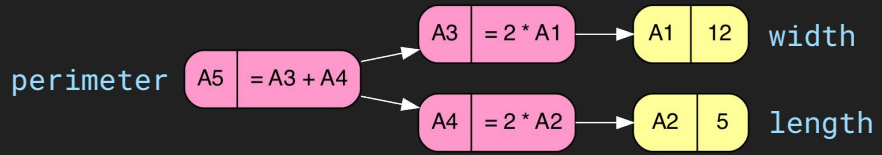
is what gives a spreadsheet its

superpower!

Many problems can be broken down into a series of smaller steps,

and these can be assigned to individual formulas in cells

```
width := 12
length := 5
perimeter := 2 * width + 2 * length
```

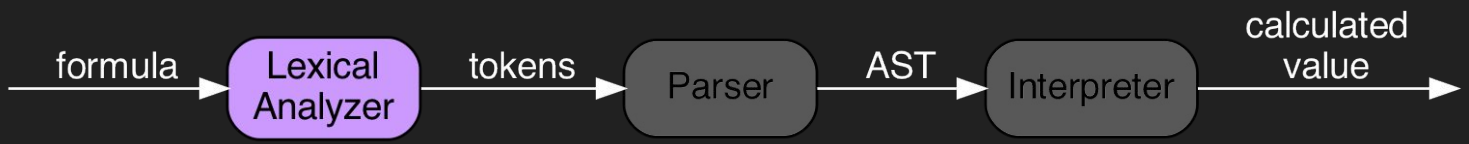


Perimeter of a rectangle

But first we need to learn how to calculate a formula...

How to Calculate Formulas

Lexical Analysis



Lexer: from Formula to Tokens

- Lexing is the process of splitting a stream of text into tokens

Lexer: from Formula to Tokens

- Lexing is the process of splitting a stream of text into tokens
- Each token has its own assigned meaning

Lexer: from Formula to Tokens

- Lexing is the process of splitting a stream of text into tokens
- Each token has its own assigned meaning
- A scanner is used to find the tokens

Lexer: from Formula to Tokens

- Lexing is the process of splitting a stream of text into tokens
- Each token has its own assigned meaning
- A scanner is used to find the tokens
- Usually based on a finite-state machine

```go
package lexer

// TokenType is the type of token found during lexing
type TokenType int

// Token holds a single token from the input string
type Token struct {
   TokenType
   Text string
}

// TokenTypes represent the different types of tokens found during lexing
const (
   TokenType_String TokenType = iota
   TokenType_Number
   TokenType_Identifier
   TokenType_Range
   TokenType_Operator
   TokenType_Comma
   TokenType_LParen
   TokenType_RParen
)
```

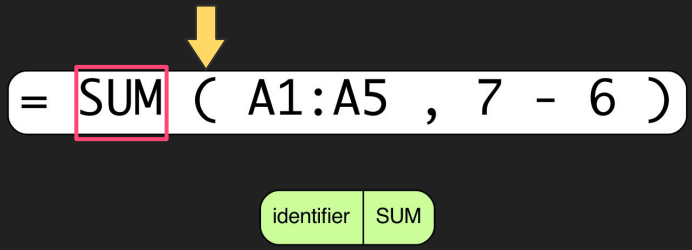
```
= SUM ( A1:A5 , 7 - 6 )
```

```
func (l *lexer) run() ([]*Token, error) {
   for {
       ch := l.next()
       if ch == '.' {
           l.lexNumber(true)
       }
       if ch == '-' || ch == '[' {
           l.lexSymbol()
       }

       if isIdentChar(ch) {
           if err := l.lexRange(); err != nil {
               if err == ErrEndOfFormula || err == ErrInvalidRange {
                   if isDigit(ch) {
                       l.lexNumber(false)
                   } else {
                       l.lexIdent()
                   }
               } else {
                   return nil, err
               }
           }
           continue
       }

       if err := l.lexSymbol(); err != nil {
           return nil, err
       }
   }
   return l.tokens, nil
}
```


```go
func (l *lexer) lexIdentifier() error {
    var ch byte
    for {
        ch = l.next()
        if isIdentifierChar(ch) {
            continue
        }

        if ch != eof {
            l.backup()
        }
        return l.emit(TokenType_Identifier)
    }
}
```

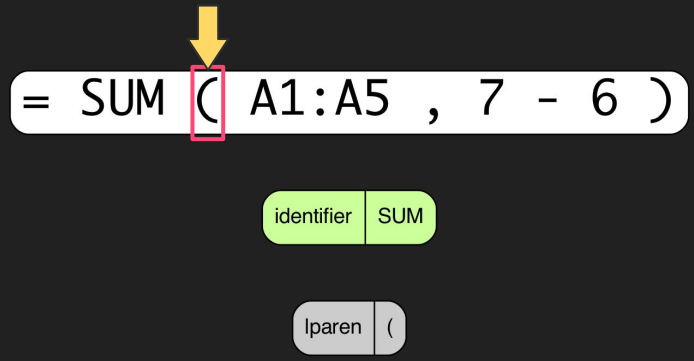

```
func (l *lexer) lexIdentifier() error {
   var ch byte
   for {
       ch = l.next()
       if isIdentifierChar(ch) {
           continue
       }

       if ch != eof {
           l.backup()
       }
       return l.emit(TokenType_Identifier)
   }
}
```


```
func (l *lexer) lexIdentifier() error {
   var ch byte
   for {
       ch = l.next()
       if isIdentifierChar(ch) {
           continue
       }

       if ch != eof {
           l.backup()
       }
       return l.emit(TokenType_Identifier)
   }
}
```

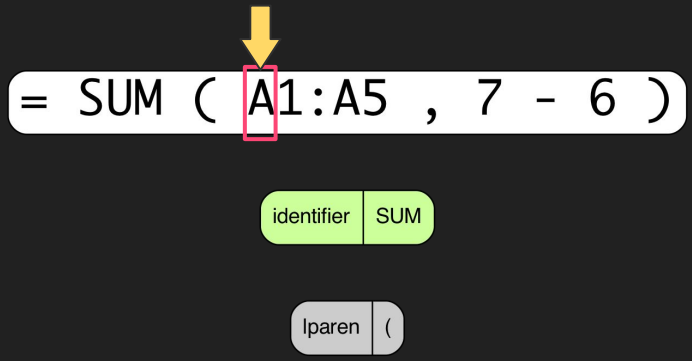



```
func (l *lexer) lexIdentifier() error {
   var ch byte
   for {
       ch = l.next()
       if isIdentifierChar(ch) {
           continue
       }

       if ch != eof {
           l.backup()
       }
       return l.emit(TokenType_Identifier)
   }
}
```

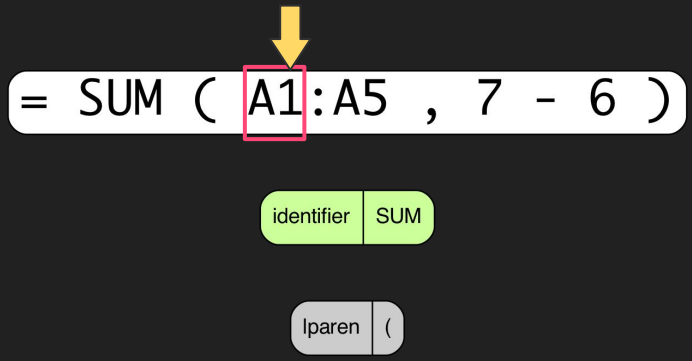
```
func (l *lexer) lexSymbol() error {
    l.backup()
    ch := l.next()
    switch ch {
    case '(':
        return l.emit(TokenType_LParen)
    case ')':
        return l.emit(TokenType_RParen)
    case '+', '-', '*', '/', '=', '&', '^', '%':
        return l.emit(TokenType_Operator)
    case ',':
        return l.emit(TokenType_Comma)
    case '"', '\'':
        return l.lexString(ch)
    case '<', '>':
        nextCh := l.next()
        if nextCh == eof ||
            nextCh == '=' ||
            (ch == '<' && nextCh == '>') {
            return l.emit(TokenType_Operator)
        }
        l.backup()
        return l.emit(TokenType_Operator)
    }
    return unexpectedCharacter(ch)
}
```



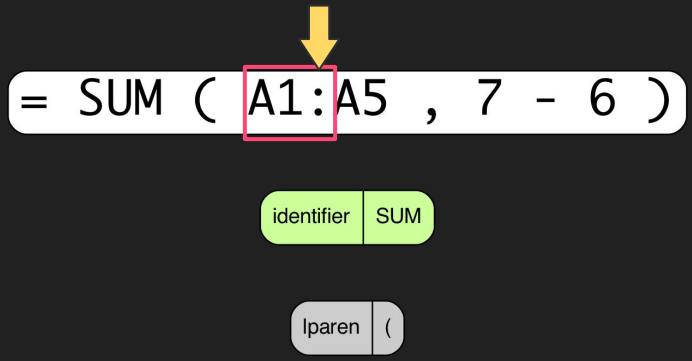
```go
func (l *lexer) lexRange() error {
    seenColon := false
    for {
        ch := l.next()
        if ch == eof {
            if seenColon {
                l.backup()
                return l.emit(TokenType_Range)
            }
            return ErrEndOfFormula
        }

        if ch == ':' {
            if seenColon {
                return ErrInvalidRange
            }
            seenColon = true
            continue
        }

        if isCellChar(ch) {
            continue
        }

        if !seenColon {
            return ErrInvalidRange
        }

        l.backup()
        return l.emit(TokenType_Range)
    }
}
```



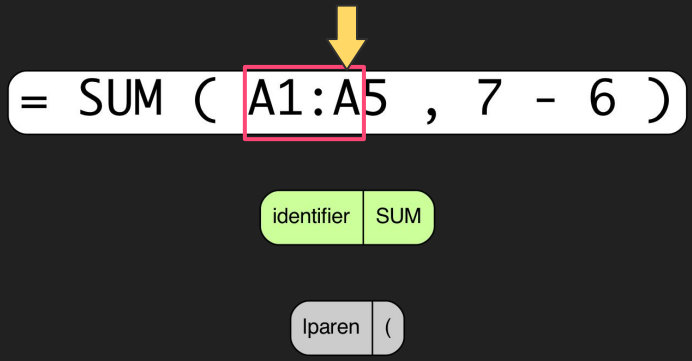
```go
func (l *lexer) lexRange() error {
    seenColon := false
    for {
        ch := l.next()
        if ch == eof {
            if seenColon {
                l.backup()
                return l.emit(TokenType_Range)
            }
            return ErrEndOfFormula
        }

        if ch == ':' {
            if seenColon {
                return ErrInvalidRange
            }
            seenColon = true
            continue
        }

        if isCellChar(ch) {
            continue
        }

        if !seenColon {
            return ErrInvalidRange
        }

        l.backup()
        return l.emit(TokenType_Range)
    }
}
```



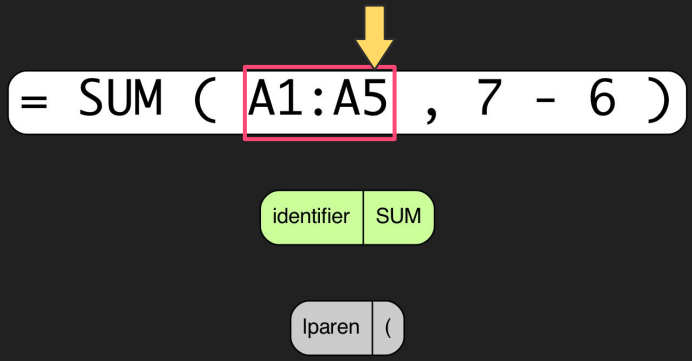
```go
func (l *lexer) lexRange() error {
    seenColon := false
    for {
        ch := l.next()
        if ch == eof {
            if seenColon {
                l.backup()
                return l.emit(TokenType_Range)
            }
            return ErrEndOfFormula
        }

        if ch == ':' {
            if seenColon {
                return ErrInvalidRange
            }
            seenColon = true
            continue
        }

        if isCellChar(ch) {
            continue
        }

        if !seenColon {
            return ErrInvalidRange
        }

        l.backup()
        return l.emit(TokenType_Range)
    }
}
```



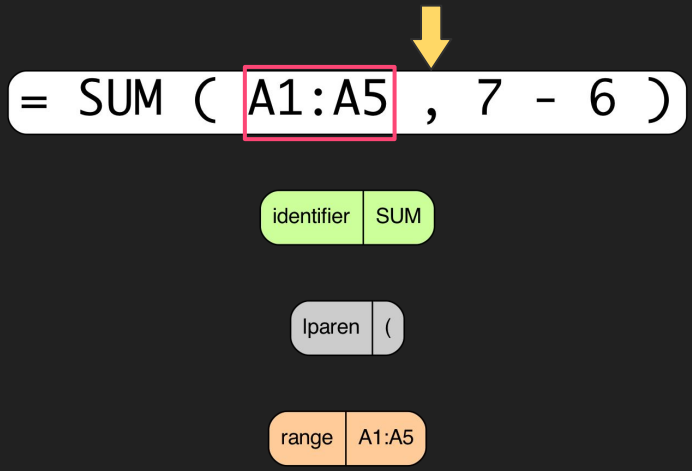
```go
func (l *lexer) lexRange() error {
    seenColon := false
    for {
        ch := l.next()
        if ch == eof {
            if seenColon {
                l.backup()
                return l.emit(TokenType_Range)
            }
            return ErrEndOfFormula
        }

        if ch == ':' {
            if seenColon {
                return ErrInvalidRange
            }
            seenColon = true
            continue
        }

        if isCellChar(ch) {
            continue
        }

        if !seenColon {
            return ErrInvalidRange
        }

        l.backup()
        return l.emit(TokenType_Range)
    }
}
```



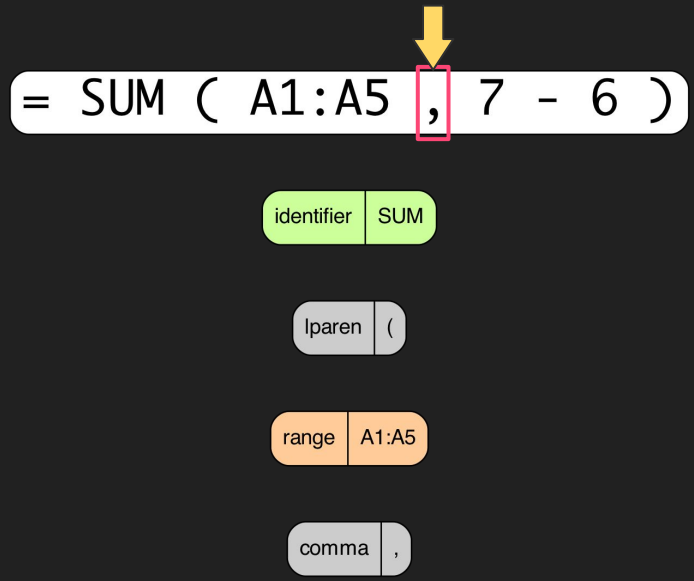
```go
func (l *lexer) lexRange() error {
    seenColon := false
    for {
        ch := l.next()
        if ch == eof {
            if seenColon {
                l.backup()
                return l.emit(TokenType_Range)
            }
            return ErrEndOfFormula
        }

        if ch == ':' {
            if seenColon {
                return ErrInvalidRange
            }
            seenColon = true
            continue
        }

        if isCellChar(ch) {
            continue
        }

        if !seenColon {
            return ErrInvalidRange
        }

        l.backup()
        return l.emit(TokenType_Range)
    }
}
```



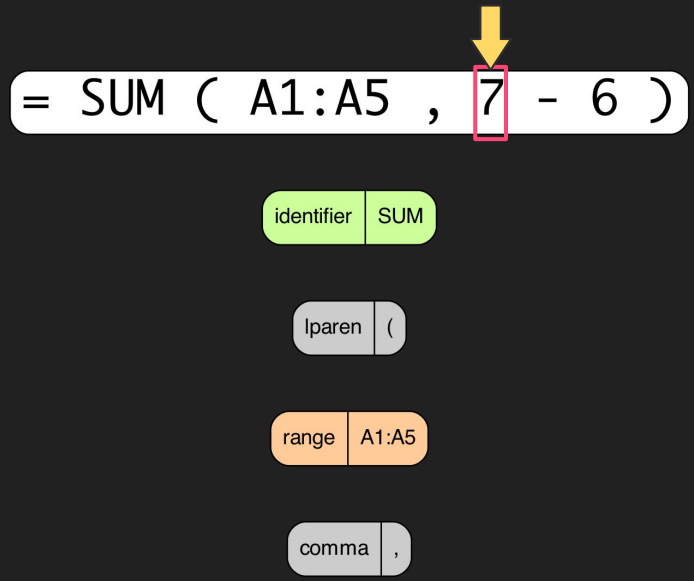
```go
func (l *lexer) lexRange() error {
    seenColon := false
    for {
        ch := l.next()
        if ch == eof {
            if seenColon {
                l.backup()
                return l.emit(TokenType_Range)
            }
            return ErrEndOfFormula
        }

        if ch == ':' {
            if seenColon {
                return ErrInvalidRange
            }
            seenColon = true
            continue
        }

        if isCellChar(ch) {
            continue
        }

        if !seenColon {
            return ErrInvalidRange
        }

        l.backup()
        return l.emit(TokenType_Range)
    }
}
```



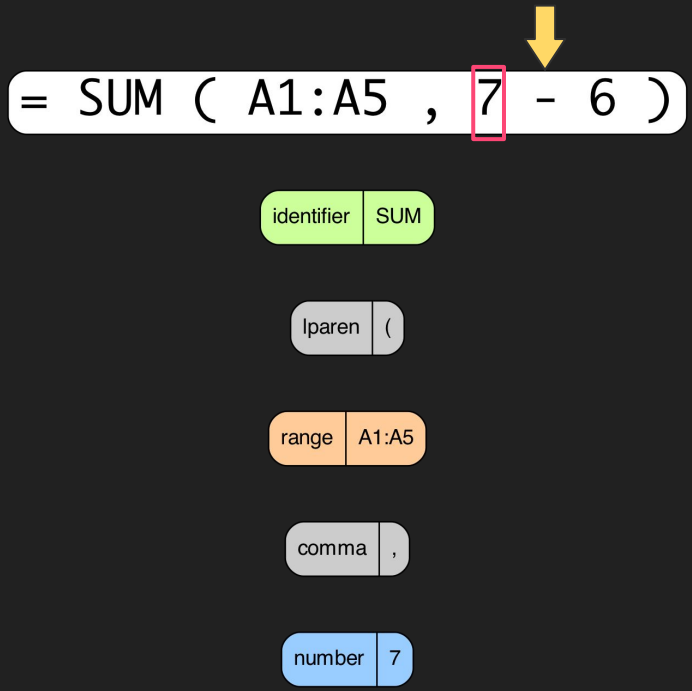
```
func (l *lexer) lexSymbol() error {
   l.backup()
   ch := l.next()
   switch ch {
   case '(':
       return l.emit(TokenType_LParen)
   case ')':
       return l.emit(TokenType_RParen)
   case '+', '-', '*', '/', '=', '&', '^', '%':
       return l.emit(TokenType_Operator)
   case ',':
       return l.emit(TokenType_Comma)
   case '"', '\'':
       return l.lexString(ch)
   case '<', '>':
       nextCh := l.next()
       if nextCh == eof ||
           nextCh == '=' ||
           (ch == '<' && nextCh == '>') {
           return l.emit(TokenType_Operator)
       }
       l.backup()
       return l.emit(TokenType_Operator)
   }
   return unexpectedCharacter(ch)
}
```



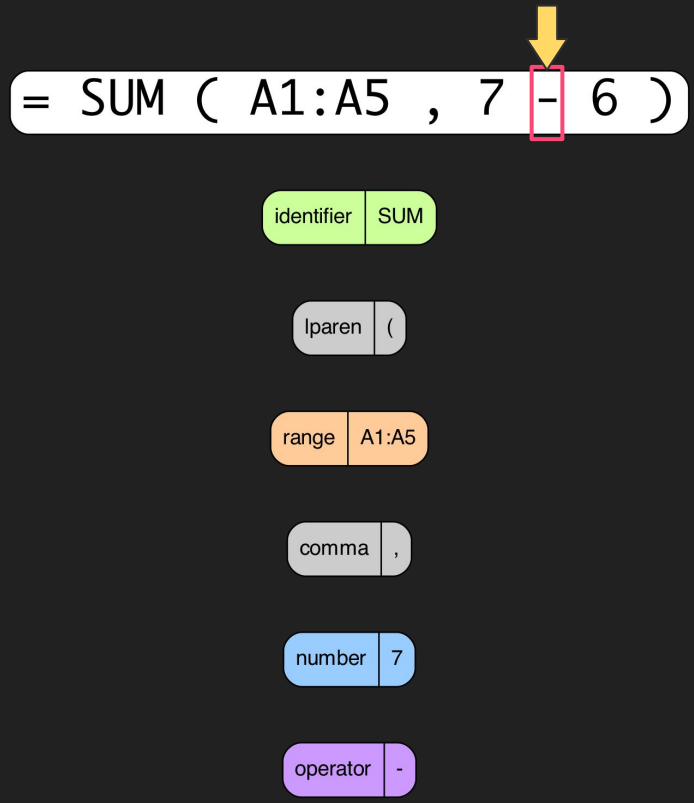
```
func (l *lexer) lexNumber(seenDecimal bool) error {
    var ch byte
    for {
        ch = l.next()

        if ch == eof {
            l.emit(TokenType_Number)
            return nil
        }

        if ch == '.' {
            if seenDecimal {
                return ErrInvalidNumber
            }
            seenDecimal = true
            continue
        }

        if isDigit(ch) {
            continue
        }

        l.backup()
        return l.emit(TokenType_Number)
    }
}
```



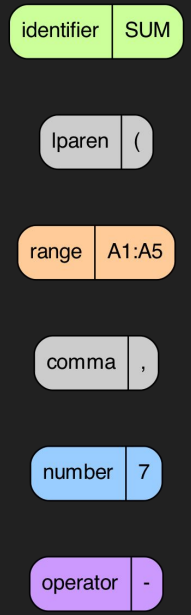
```
func (l *lexer) lexNumber(seenDecimal bool) error {
    var ch byte
    for {
        ch = l.next()

        if ch == eof {
            l.emit(TokenType_Number)
            return nil
        }

        if ch == '.' {
            if seenDecimal {
                return ErrInvalidNumber
            }
            seenDecimal = true
            continue
        }

        if isDigit(ch) {
            continue
        }

        l.backup()
        return l.emit(TokenType_Number)
    }
}
```



```go
func (l *lexer) lexSymbol() error {
    l.backup()
    ch := l.next()
    switch ch {
    case '(':
        return l.emit(TokenType_LParen)
    case ')':
        return l.emit(TokenType_RParen)
    case '+', '-', '*', '/', '=', '&', '^', '%':
        return l.emit(TokenType_Operator)
    case ',':
        return l.emit(TokenType_Comma)
    case '"', '\'':
        return l.lexString(ch)
    case '<', '>':
        nextCh := l.next()
        if nextCh == eof ||
            nextCh == '=' ||
            (ch == '<' && nextCh == '>') {
            return l.emit(TokenType_Operator)
        }
        l.backup()
        return l.emit(TokenType_Operator)
    }
    return unexpectedCharacter(ch)
}
```




```go
func (l *lexer) lexNumber(seenDecimal bool) error {
    var ch byte
    for {
        ch = l.next()

        if ch == eof {
            l.emit(TokenType_Number)
            return nil
        }

        if ch == '.' {
            if seenDecimal {
                return ErrInvalidNumber
            }
            seenDecimal = true
            continue
        }

        if isDigit(ch) {
            continue
        }

        l.backup()
        return l.emit(TokenType_Number)
    }
}
```

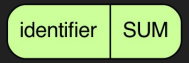


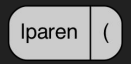
```
= SUM ( A1:A5 , 7 - 6 )
```

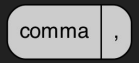
```go
func (l *lexer) lexNumber(seenDecimal bool) error {
    var ch byte
    for {
        ch = l.next()

        if ch == eof {
            l.emit(TokenType_Number)
            return nil
        }

        if ch == '.' {
            if seenDecimal {
                return ErrInvalidNumber
            }
            seenDecimal = true
            continue
        }

        if isDigit(ch) {
            continue
        }

        l.backup()
        return l.emit(TokenType_Number)
    }
}
```

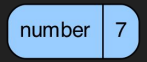
identifier	SUM
lparen	(
range	A1:A5
comma	,
number	7
operator	-
number	6

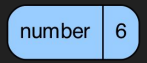
```
func (l *lexer) lexSymbol() error {
    l.backup()
    ch := l.next()
    switch ch {
    case '(':
        return l.emit(TokenType_LParen)
    case ')':
        return l.emit(TokenType_RParen)
    case '+', '-', '*', '/', '=', '&', '^', '%':
        return l.emit(TokenType_Operator)
    case ',':
        return l.emit(TokenType_Comma)
    case '"', '\'':
        return l.lexString(ch)
    case '<', '>':
        nextCh := l.next()
        if nextCh == eof ||
            nextCh == '=' ||
            (ch == '<' && nextCh == '>') {
            return l.emit(TokenType_Operator)
        }
        l.backup()
        return l.emit(TokenType_Operator)
    }
    return unexpectedCharacter(ch)
}
```

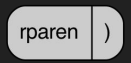


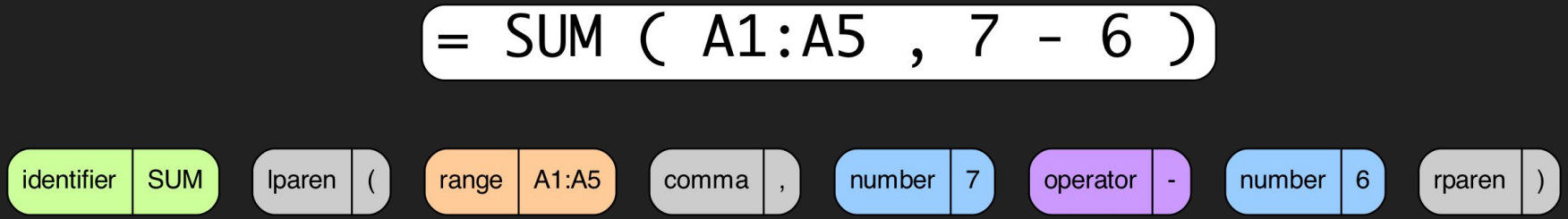




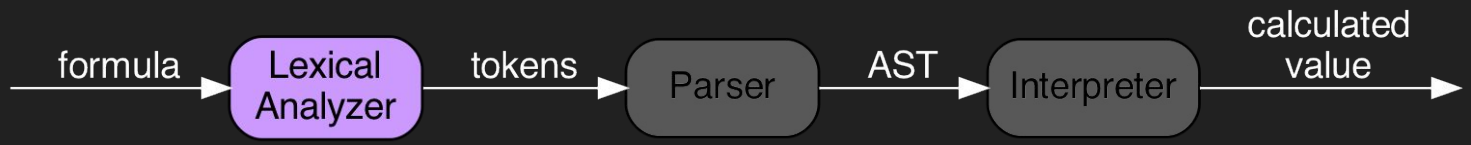







= SUM (A1:A5 , 7 - 6)
identifier
SUM
lparen
(
range
A1:A5
comma
,
number
7
operator
-
number
6
rparen
)

We’ve got tokens, now what?



Parsing Tokens to AST

Parsing Tokens to AST

- Parsing is the process of analyzing tokens to form a data structure

Parsing Tokens to AST

- Parsing is the process of analyzing tokens to form a data structure
- The input stream is checked for correct syntax

Parsing Tokens to AST

- Parsing is the process of analyzing tokens to form a data structure
- The input stream is checked for correct syntax
- Rules are defined as a context-free grammar

Parsing Tokens to AST

- Parsing is the process of analyzing tokens to form a data structure
- The input stream is checked for correct syntax
- Rules are defined as a context-free grammar

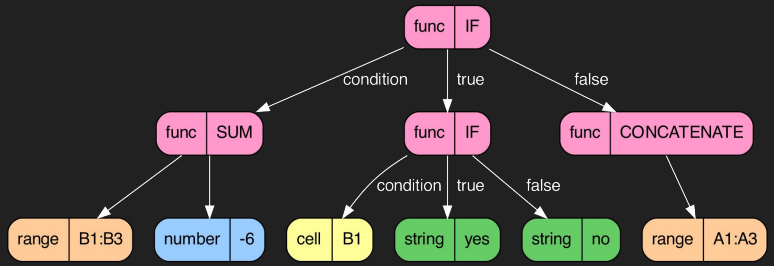
```
<Formula> ::= <Constant> | <Reference> | <FunctionCall> | ...

<FunctionCall> ::= <Identifier> '(' <Arguments> ')'

<Arguments> ::= <Argument> | <Argument> ',' <Arguments>

<Argument> ::= <Formula> | empty
```

Parsing Tokens to AST

- Parsing is the process of analyzing tokens to form a data structure
- The input stream is checked for correct syntax
- Rules are defined as a context-free grammar
- An abstract syntax tree (AST) is built to represent the formula

Parsing Tokens to AST

- Parsing is the process of analyzing tokens to form a data structure
- The input stream is checked for correct syntax
- Rules are defined as a context-free grammar
- An abstract syntax tree (AST) is built to represent the formula

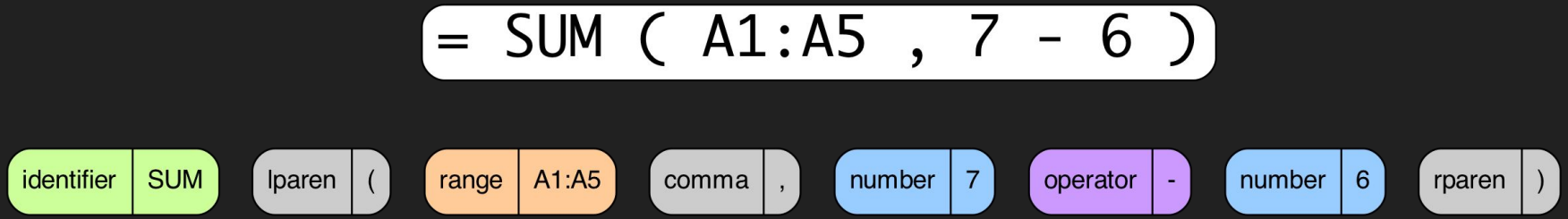


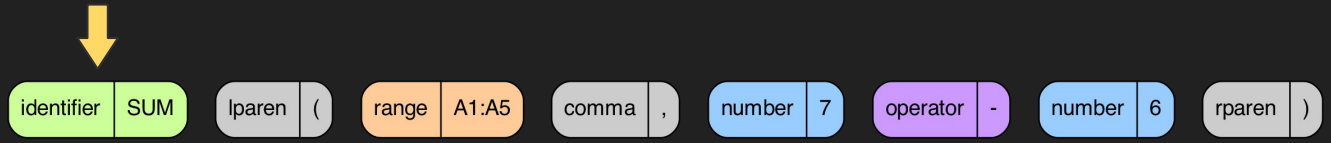
```
package parser

type (
    Node interface{}

    Operator struct {
        LHS       Node
        Operation string
        RHS       Node
    }

    Function struct {
        Name      string
        Arguments []Node
    }

    Bool struct {
        Value string
    }

    Number struct {
        Value string
    }

    String struct {
        Value string
    }

    Cell struct {
        Value string
    }

    Range struct {
        Value string
    }
```

```
package parser

type (
    Node interface{}

    Operator struct {
        LHS       Node
        Operation string
        RHS       Node
    }

    Function struct {
        Name      string
        Arguments []Node
    }
```

```
Bool struct {
    Value string
}

Number struct {
    Value string
}

String struct {
    Value string
}

Cell struct {
    Value string
}

Range struct {
    Value string
}
```

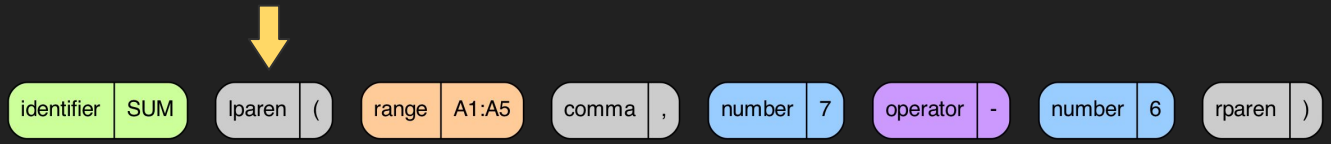
```
package parser

type (
    Node interface{}

    Operator struct {
        LHS       Node
        Operation string
        RHS       Node
    }

    Function struct {
        Name      string
        Arguments []Node
    }

    Bool struct {
        Value string
    }

    Number struct {
        Value string
    }

    String struct {
        Value string
    }

    Cell struct {
        Value string
    }

    Range struct {
        Value string
    }
```


= SUM (A1:A5 , 7 - 6)
identifier
SUM
lparen
(
range
A1:A5
comma
,
number
7
operator
-
number
6
rparen
)

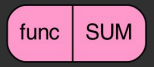


AST Output

```
switch token.TokenType {

case lexer.TokenType_Identifier:
    upper := strings.ToUpper(token.Text)
    if upper == "TRUE" || upper == "FALSE" {
        return &Bool{upper}, nil
    }

    // look for function call
    next := p.next()
    if next != nil && next.TokenType == lexer.TokenType_LParen {
        return p.function(token.Text)
    }

    return &Cell{token.Text}, nil
```



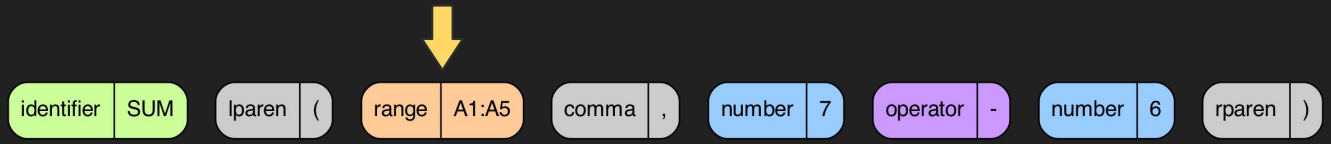
AST Output




```
switch token.TokenType {

case lexer.TokenType_Identifier:
    upper := strings.ToUpper(token.Text)
    if upper == "TRUE" || upper == "FALSE" {
        return &Bool{upper}, nil
    }

    // look for function call
    next := p.next()
    if next != nil && next.TokenType == lexer.TokenType_LParen {
        return p.function(token.Text)
    }

    return &Cell{token.Text}, nil
```

Grammar

<FunctionCall> ::= <Identifier> '(' <Arguments> ')'

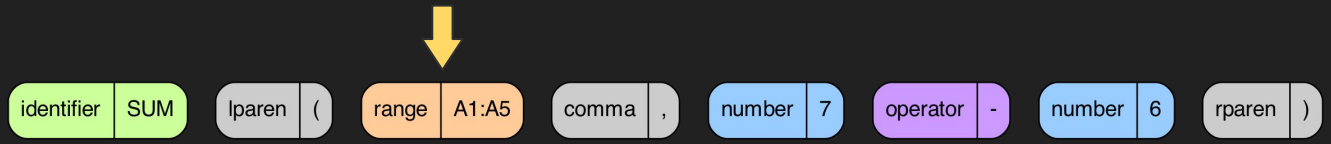


AST Output

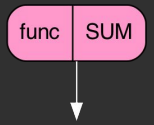
func | SUM

```go
func (p *parser) function(name string) (Node, error) {
    next := p.peek()
    if next.TokenType == lexer.TokenType_RParen {
        // 0-argument function call
        return &Function{
            Name:      name,
            Arguments: []Node{},
        }, nil
    }

    firstArgument, err := p.begin()
    if err != nil {
        return nil, err
    }

    node := &Function{
        Name:      name,
        Arguments: []Node{firstArgument},
    }

    ...
```

```go
...

for {
    next := p.next()
    switch next.TokenType {
    case lexer.TokenType_RParen:
        return node, nil
    case lexer.TokenType_Comma:
        nextToken := p.peek()
        if nextToken.TokenType == lexer.TokenType_RParen {
            arg, err = nil, nil
        } else {
            arg, err = p.begin()
        }

        // add argument to function node
        node.Arguments = append(node.Arguments, arg)
    default:
        return nil, ErrUnexpectedToken
    }
}
```

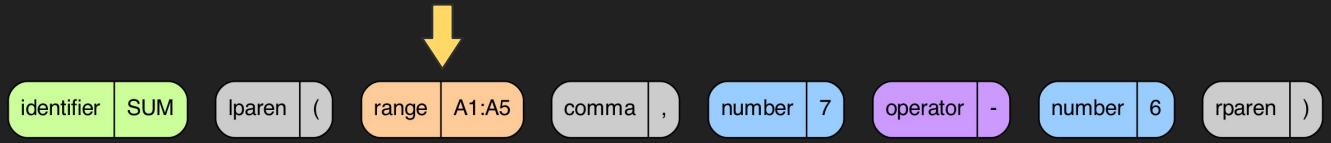


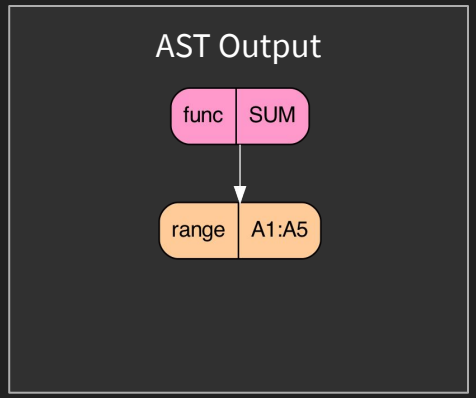
AST Output



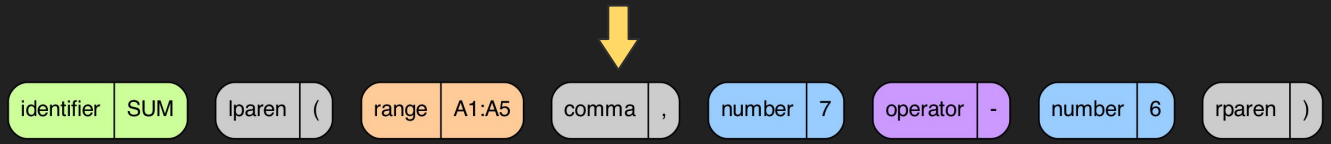

```go
func (p *parser) function(name string) (Node, error) {

    next := p.peek()
    if next.TokenType == lexer.TokenType_RParen {
        // 0-argument function call
        return &Function{
            Name:      name,
            Arguments: []Node{},
        }, nil
    }

    firstArgument, err := p.begin()
    if err != nil {
        return nil, err
    }

    node := &Function{
        Name:      name,
        Arguments: []Node{firstArgument},
    }

    ...
```

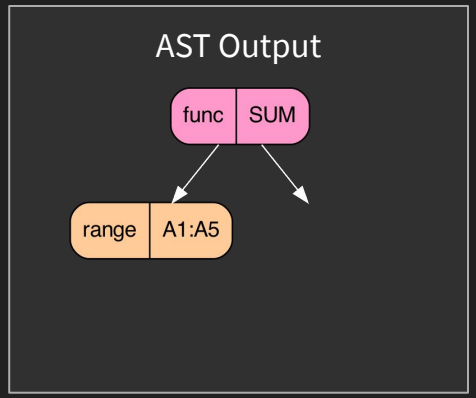
```go
...

for {
    next := p.next()
    switch next.TokenType {
    case lexer.TokenType_RParen:
        return node, nil
    case lexer.TokenType_Comma:
        nextToken := p.peek()
        if nextToken.TokenType == lexer.TokenType_RParen {
            arg, err = nil, nil
        } else {
            arg, err = p.begin()
        }

        // add argument to function node
        node.Arguments = append(node.Arguments, arg)
    default:
        return nil, ErrUnexpectedToken
    }
}
```






```
switch token.TokenType {

case lexer.TokenType_Number:
    return &Number{token.Text}, nil

case lexer.TokenType_String:
    return &String{token.Text}, nil

case lexer.TokenType_Range:
    return &Range{token.Text}, nil
```



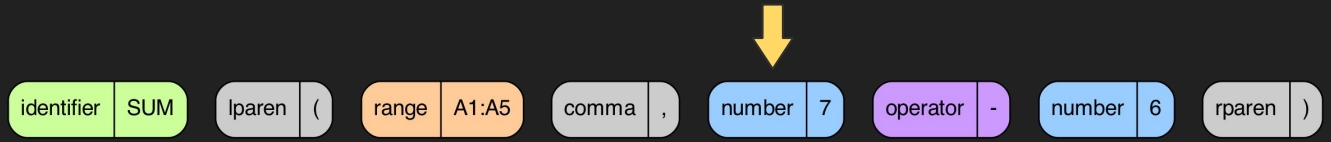


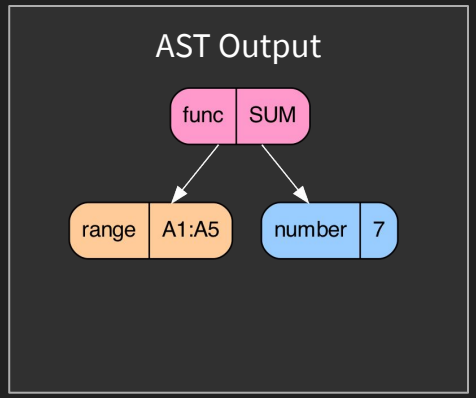

```go
func (p *parser) function(name string) (Node, error) {

    next := p.peek()
    if next.TokenType == lexer.TokenType_RParen {
        // 0-argument function call
        return &Function{
            Name:      name,
            Arguments: []Node{},
        }, nil
    }

    firstArgument, err := p.begin()
    if err != nil {
        return nil, err
    }

    node := &Function{
        Name:      name,
        Arguments: []Node{firstArgument},
    }

    ...
```

```go
...

for {
    next := p.next()
    switch next.TokenType {
    case lexer.TokenType_RParen:
        return node, nil
    case lexer.TokenType_Comma:
        nextToken := p.peek()
        if nextToken.TokenType == lexer.TokenType_RParen {
            arg, err = nil, nil
        } else {
            arg, err = p.begin()
        }

        // add argument to function node
        node.Arguments = append(node.Arguments, arg)
    default:
        return nil, ErrUnexpectedToken
    }
}
```

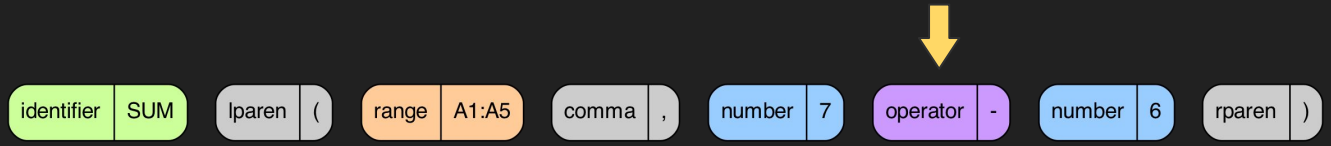
Grammar

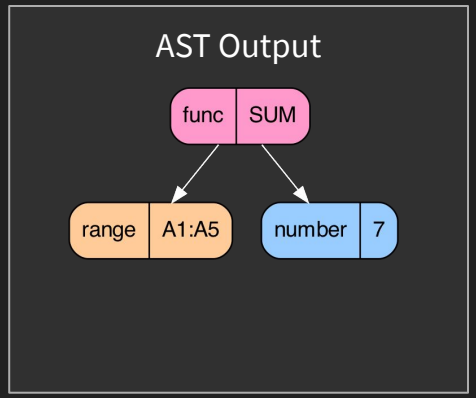
```
<Arguments> ::= <Argument> | <Argument> ',' <Arguments>
```



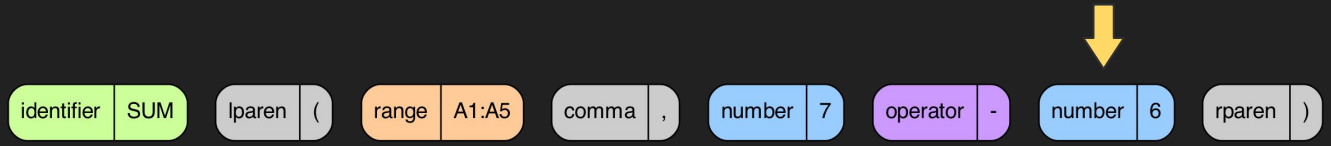


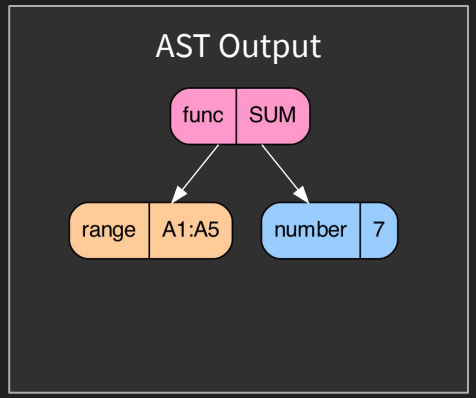

```
switch token.TokenType {

case lexer.TokenType_Number:
    return &Number{token.Text}, nil

case lexer.TokenType_String:
    return &String{token.Text}, nil

case lexer.TokenType_Range:
    return &Range{token.Text}, nil
```



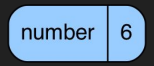


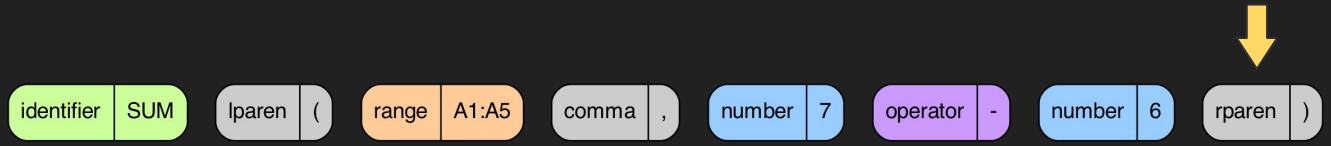

```go
lhs, err := buildSubTree()
if err != nil {
    return nil, err
}

operatorToken := p.next()
if operatorToken == nil {
    return lhs, nil
}

if operatorToken.TokenType != lexer.TokenType_Operator {
    p.backup()
    return lhs, nil
}
```

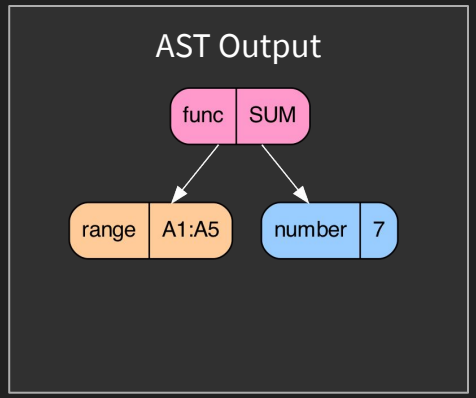




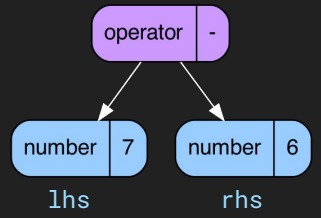

```
for {
    rhs, err := buildSubTree()
    if err != nil {
        return nil, err
    }

    lhs = &Operator{
        lhs,
        operatorToken.Text,
        rhs,
    }

    operatorToken = p.next()
    if operatorToken == nil {
        return lhs, nil
    }

    if operatorToken.TokenType != lexer.TokenType_Operator {
        p.backup()
        return lhs, nil
    }
}
```

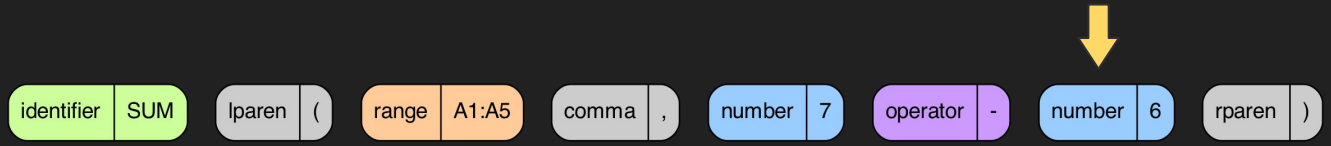


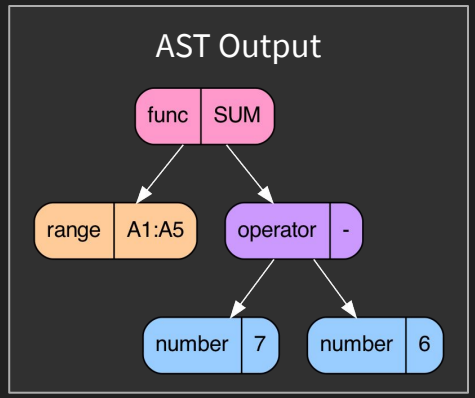




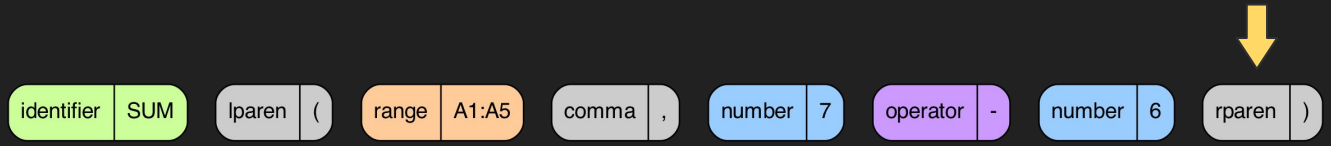

```
for {
    rhs, err := buildSubTree()
    if err != nil {
        return nil, err
    }

    lhs = &Operator{
        lhs,
        operatorToken.Text,
        rhs,
    }

    operatorToken = p.next()
    if operatorToken == nil {
        return lhs, nil
    }

    if operatorToken.TokenType != lexer.TokenType_Operator {
        p.backup()
        return lhs, nil
    }
}
```

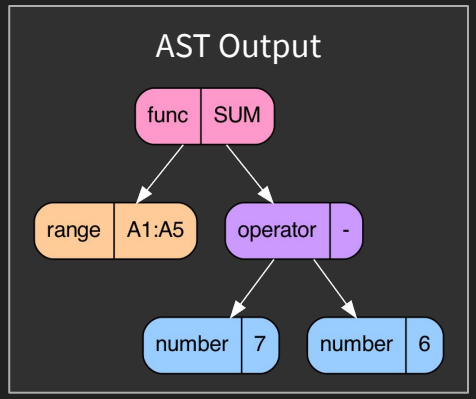







```
for {
    rhs, err := buildSubTree()
    if err != nil {
        return nil, err
    }

    lhs = &Operator{
        lhs,
        operatorToken.Text,
        rhs,
    }

    operatorToken = p.next()
    if operatorToken == nil {
        return lhs, nil
    }

    if operatorToken.TokenType != lexer.TokenType_Operator {
        p.backup()
        return lhs, nil
    }
}
```

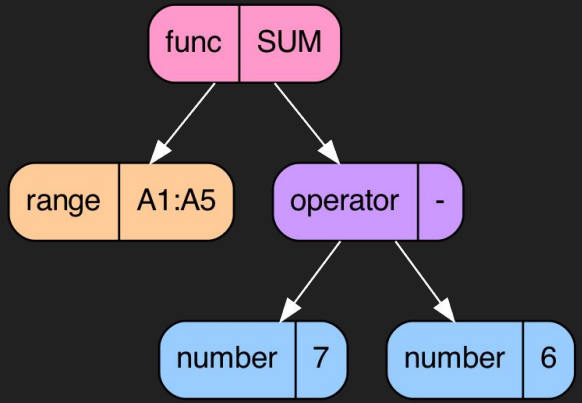


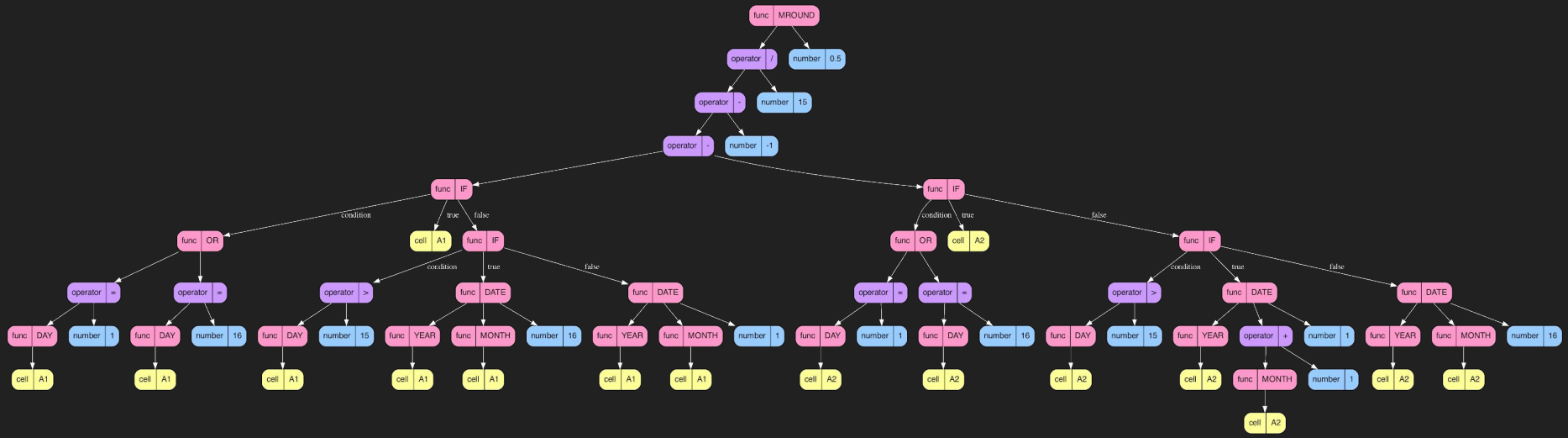


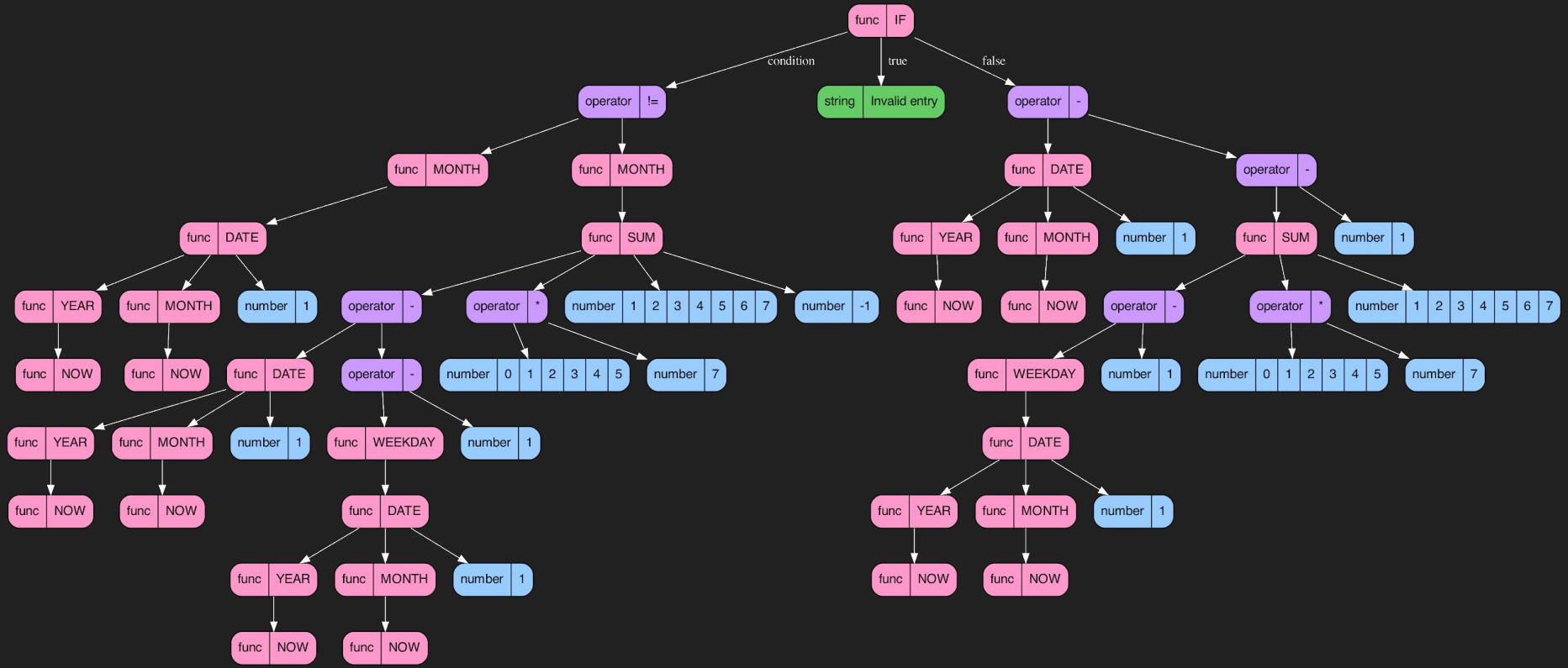

```go
func (p *parser) function(name string) (Node, error) {

    next := p.peek()
    if next.TokenType == lexer.TokenType_RParen {
        // 0-argument function call
        return &Function{
            Name:      name,
            Arguments: []Node{},
        }, nil
    }

    firstArgument, err := p.begin()
    if err != nil {
        return nil, err
    }

    node := &Function{
        Name:      name,
        Arguments: []Node{firstArgument},
    }

    ...
```

```go
...

for {
    next := p.next()
    switch next.TokenType {
    case lexer.TokenType_RParen:
        return node, nil
    case lexer.TokenType_Comma:
        nextToken := p.peek()
        if nextToken.TokenType == lexer.TokenType_RParen {
            arg, err = nil, nil
        } else {
            arg, err = p.begin()
        }

        // add argument to function node
        node.Arguments = append(node.Arguments, arg)
    default:
        return nil, ErrUnexpectedToken
    }
}
```

```
= SUM ( A1:A5 , 7 - 6 )
```

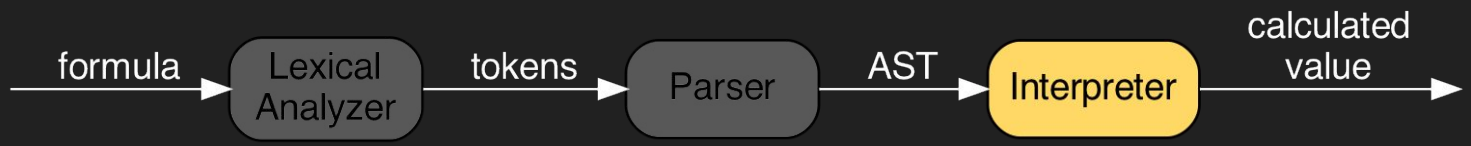



func MROUND
operator /
number 0.5
operator -
number 15
operator +
number -1
func IF
condition
true
false
func OR
operator =
func DAY
number 1
number 16
cell A1
operator >
number 15
func DATE
func YEAR
func MONTH
cell A2
operator +


func IF
condition
true
false
operator !=
string Invalid entry
operator -
func MONTH
func MONTH
func DATE
func DATE
operator -
func SUM
func YEAR
func MONTH
number 1
func YEAR
func MONTH
number 1
operator -
operator *
number 1 2 3 4 5 6 7
number -1
func NOW
func NOW
func NOW
func NOW
func DATE
operator -
number 0 1 2 3 4 5
number 7
operator -
operator *
number 1 2 3 4 5 6 7
func YEAR
func MONTH
number 1
func WEEKDAY
number 1
func WEEKDAY
number 1
number 0 1 2 3 4 5
number 7
func NOW
func NOW
func DATE
func DATE
func YEAR
func MONTH
number 1
func YEAR
func MONTH
number 1
func NOW
func NOW
func NOW
func NOW

Now for the Final Step

Let’s Calculate our Formula!



	A	B
1	1	= SUM(A1:A5 , 7-6)
2	2	
3	3	
4	4	
5	5	

```
engine := calc.NewEngine("simple")
sheet := calc.NewSheet(10, 10, engine)

sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5, 7 - 6)`)

sheet.Calculate()
```

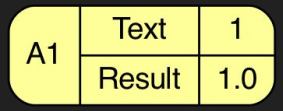
```
type CalculationEngine interface {
   Parse(row, col int, ctx *Context)
   Calculate(row, col int, ctx *Context) *Result
}
```

```
type Cell struct {
   Row, Col  int         // The location of this cell
   Text      string      // The display text of this cell
   *Result               // The result of this cell

   IsFormula bool        // Is this cell value a formula?
   AST       parser.Node // The formula parsed into an abstract syntax tree (AST)
}
```

```go
type ResultType int

const (
    ResultType_Empty ResultType = iota
    ResultType_Bool
    ResultType_Decimal
    ResultType_String
    ResultType_Range
)

// Result is a union struct of num|text|ref
type Result struct {
    Type ResultType // the type of this result
    Dec  float64    // number representation
    Str  string     // text representation
    Ref  Range      // cell or range reference
}
```

```
sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5, 7 - 6)`)

sheet.Calculate()
```

```go
func (s *Sheet) SetCell(ref, val string) {
    row, col := convert.ToRowCol(ref)
    cell := s.cells.Set(row, col, val)

    if cell.IsFormula {
        s.engine.Parse(row, col, s.cells) // parse to an AST
    } else {
        s.cells.SetResult(row, col, result.FromText(val))
    }
}
```




```go
func FromText(text string) *model.Result {
    upper := strings.ToUpper(text)
    if upper == "TRUE" || upper == "FALSE" {
        return Bool(upper == "TRUE") // ResultType_Bool
    }

    if ival, err := strconv.Atoi(text); err == nil {
        return Decimal(float64(ival)) // ResultType_Decimal
    }

    return String(text) // ResultType_String
}
```

A1	Text	1
	Result	1.0

A2	Text	2
	Result	2.0

```
sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5, 7 - 6)`)

sheet.Calculate()
```

A1	Text	1
	Result	1.0

A2	Text	2
	Result	2.0

A3	Text	3
	Result	3.0

```
sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5, 7 - 6)`)

sheet.Calculate()
```

A1	Text	1
	Result	1.0

A2	Text	2
	Result	2.0

A3	Text	3
	Result	3.0

A4	Text	4
	Result	4.0

```
sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5, 7 - 6)`)

sheet.Calculate()
```

A1	Text	1
	Result	1.0

A2	Text	2
	Result	2.0

A3	Text	3
	Result	3.0

A4	Text	4
	Result	4.0

A5	Text	5
	Result	5.0

```
sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5, 7 - 6)`)

sheet.Calculate()
```

A1	Text	1
	Result	1.0

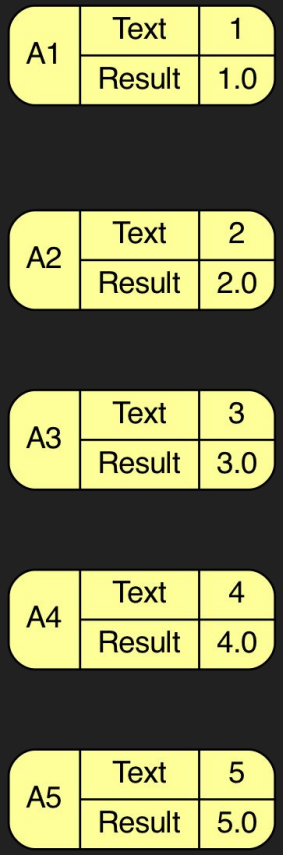
A2	Text	2
	Result	2.0

A3	Text	3
	Result	3.0

A4	Text	4
	Result	4.0

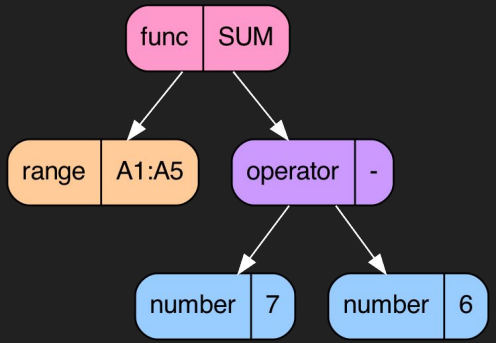
A5	Text	5
	Result	5.0

```
sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5, 7 - 6)`)

sheet.Calculate()
```




```go
func (s *Sheet) SetCell(ref, val string) {
    row, col := convert.ToRowCol(ref)
    cell := s.cells.Set(row, col, val)

    if cell.IsFormula {
        s.engine.Parse(row, col, s.cells) // parse to an AST
    } else {
        s.cells.SetResult(row, col, result.FromText(val))
    }
}
```

```
func (eng *simpleEngine) Parse(row, col int, ctx *model.Context) {
   cell := ctx.Cells.GetCell(row, col)
   cell.ParseAST()
}
```



A1		
	Text	1
	Result	1.0

B1		
	Text	=SUM(A1:A5, 7 - 6)
	Result	empty
	AST	*parser.Node

A2		
	Text	2
	Result	2.0

A3		
	Text	3
	Result	3.0

A4		
	Text	4
	Result	4.0

A5		
	Text	5
	Result	5.0

```go
func (s *Sheet) SetCell(ref, val string) {
    row, col := convert.ToRowCol(ref)
    cell := s.cells.Set(row, col, val)

    if cell.IsFormula {
        s.engine.Parse(row, col, s.cells) // parse to an AST
    } else {
        s.cells.SetResult(row, col, result.FromText(val))
    }
}
```

A1	Text	1
	Result	1.0

B1	Text	=SUM(A1:A5, 7 - 6)
	Result	empty
	AST	*parser.Node

A2	Text	2
	Result	2.0

A3	Text	3
	Result	3.0

A4	Text	4
	Result	4.0

A5	Text	5
	Result	5.0

```
sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5, 7 - 6)`)

sheet.Calculate()
```

```go
func (eng *simpleEngine) Calculate(row, col int, ctx *model.Context) *model.Result {
    cell := ctx.Cells.GetCell(row, col)
    if !cell.IsFormula {
        return cell.Result
    }
    if cell.Result.Type != model.ResultType_Empty {
        return cell.Result
    }

    cell.ParseAST()
    result := calculateNode(cell.AST, ctx)
    cell.SetResult(result)
    return result
}
```

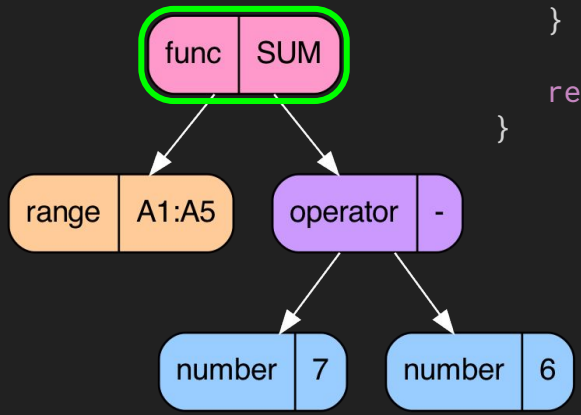
We use Postorder* Tree Traversal to calculate the AST

Postorder Algorithm

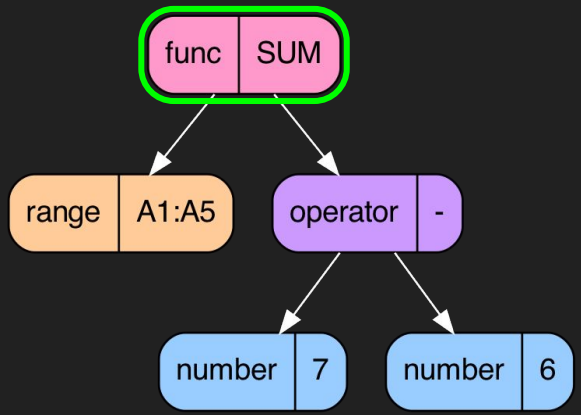
1. Traverse the left subtree
2. Traverse the right subtree(s)
3. Visit the root

a.k.a. RPN (Reverse Polish Notation)

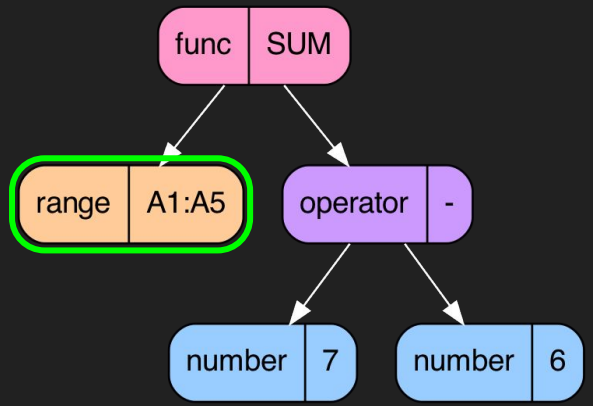
```go
func calculateNode(node parser.Node, ctx *model.Context) (r *model.Result) {
    switch n := node.(type) {
    case *parser.Bool:
        return result.Bool(n.Value == `TRUE`)

    case *parser.Number:
        f, _ := strconv.ParseFloat(n.Value, 64)
        return result.Decimal(f)

    case *parser.String:
        return result.String(n.Value)
...

    }

    return model.EmptyResult
}
```



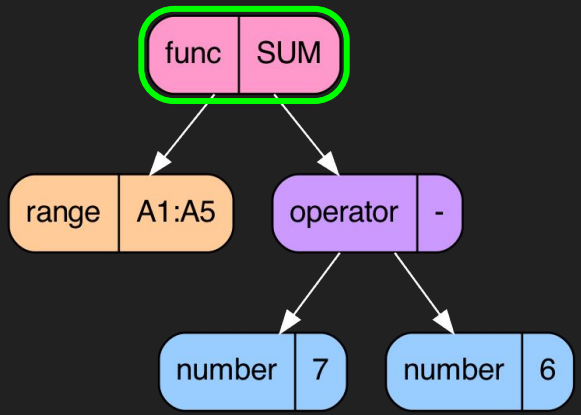
```
func calculateNode(node parser.Node, ctx *model.Context) (r *model.Result) {
   switch n := node.(type) {
...
   case *parser.Function:
       // gather the function arguments as results
       args := make([]*model.Result, len(n.Arguments))
       for i, arg := range n.Arguments {
           args[i] = calculateNode(arg, ctx)
       }

       return functions.CallByName(n.Name, ctx, args...)
```



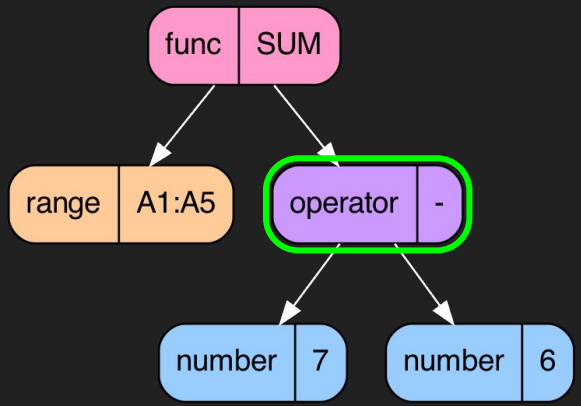
```
func calculateNode(node parser.Node, ctx *model.Context) (r *model.Result) {
   switch n := node.(type) {
...
   case *parser.Range:
       rng, err := convert.ToRange(n.Value, false)
       if err != nil {
           panic(err)
       }

       // let's pre-calculate ALL the cells in this range
       ctx.Cells.ForEach(&rng, func(cell *model.Cell) {
           ctx.Engine.Calculate(cell.Row, cell.Col, ctx)
       })

       return result.Range(rng)
```



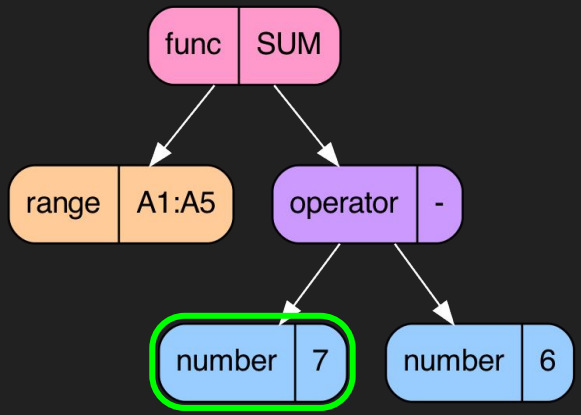
```
func calculateNode(node parser.Node, ctx *model.Context) (r *model.Result) {
   switch n := node.(type) {
...
   case *parser.Function:
       // gather the function arguments as results
       args := make([]*model.Result, len(n.Arguments))
       for i, arg := range n.Arguments {
           args[i] = calculateNode(arg, ctx)
       }

       return functions.CallByName(n.Name, ctx, args...)
```



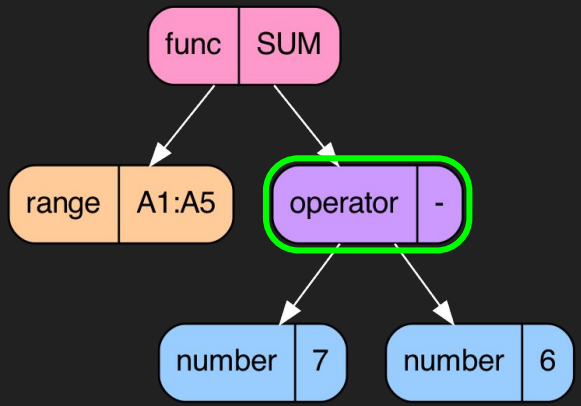
```go
func calculateNode(node parser.Node, ctx *model.Context) (r *model.Result) {
    switch n := node.(type) {
...
    case *parser.Operator:
        lResult := &model.Result{}
        if n.LHS != nil {
            lResult = calculateNode(n.LHS, ctx)
        }

        rResult := &model.Result{}
        if n.RHS != nil {
            rResult = calculateNode(n.RHS, ctx)
        }

        funcName := functions.OperatorNames(n.Operation) // "-" → "SUB"
        return functions.CallByName(funcName, ctx, lResult, rResult)
```



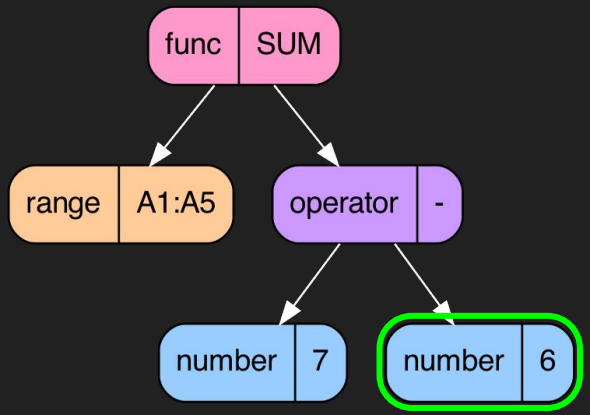
```
func calculateNode(node parser.Node, ctx *model.Context) (r *model.Result) {
   switch n := node.(type) {
...
   case *parser.Number:
       f, _ := strconv.ParseFloat(n.Value, 64)
       return result.Decimal(f)
```



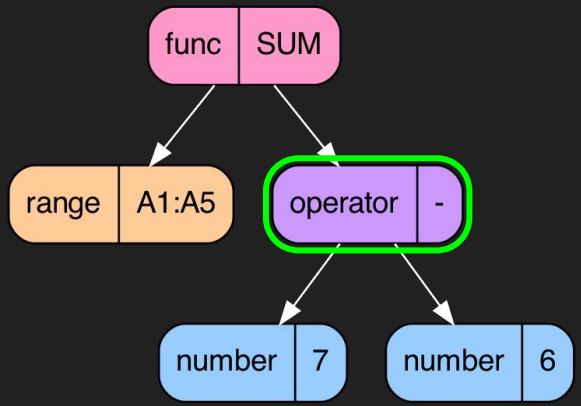
```go
func calculateNode(node parser.Node, ctx *model.Context) (r *model.Result) {
   switch n := node.(type) {
...
   case *parser.Operator:
       lResult := &model.Result{}
       if n.LHS != nil {
           lResult = calculateNode(n.LHS, ctx)
       }

       rResult := &model.Result{}
       if n.RHS != nil {
           rResult = calculateNode(n.RHS, ctx)
       }

       funcName := functions.OperatorNames(n.Operation) // "-" → "SUB"
       return functions.CallByName(funcName, ctx, lResult, rResult)
```



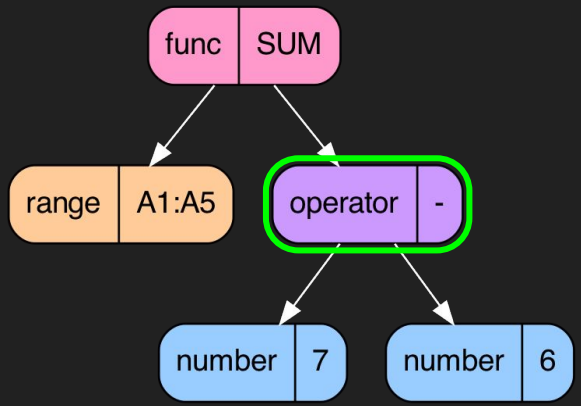
```
func calculateNode(node parser.Node, ctx *model.Context) (r *model.Result) {
   switch n := node.(type) {
...
   case *parser.Number:
       f, _ := strconv.ParseFloat(n.Value, 64)
       return result.Decimal(f)
```



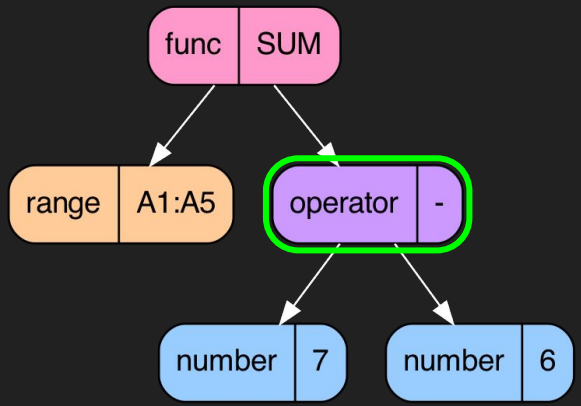
```go
func calculateNode(node parser.Node, ctx *model.Context) (r *model.Result) {
    switch n := node.(type) {
...
    case *parser.Operator:
        lResult := &model.Result{}
        if n.LHS != nil {
            lResult = calculateNode(n.LHS, ctx)
        }

        rResult := &model.Result{}
        if n.RHS != nil {
            rResult = calculateNode(n.RHS, ctx)
        }

        funcName := functions.OperatorNames(n.Operation) // "-" → "SUB"
        return ctx.Functions.CallByName(funcName, lResult, rResult)
```



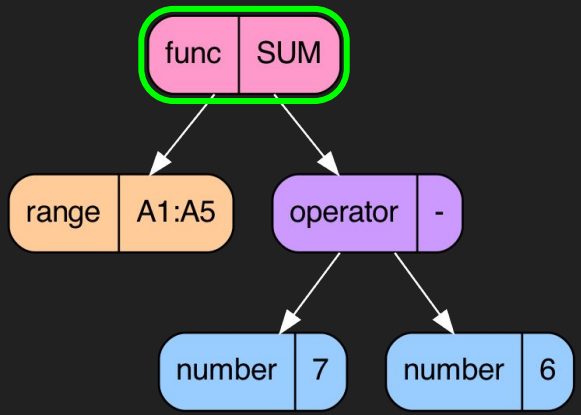
```
type Functions interface {
   CallByID(id int, args ...*Result) *Result
   CallByName(name string, args ...*Result) *Result

   Val(r, c int) *Result
   Vals(r1, c1, r2, c2 int) []*Result

   Add(args ...*Result) *Result
   Sub(args ...*Result) *Result
   Mul(args ...*Result) *Result
   Div(args ...*Result) *Result

   Concatenate(args ...*Result) *Result
   Sum(args ...*Result) *Result
   Indirect(args ...*Result) *Result

   // all of the other common spreadsheet functions
...
```



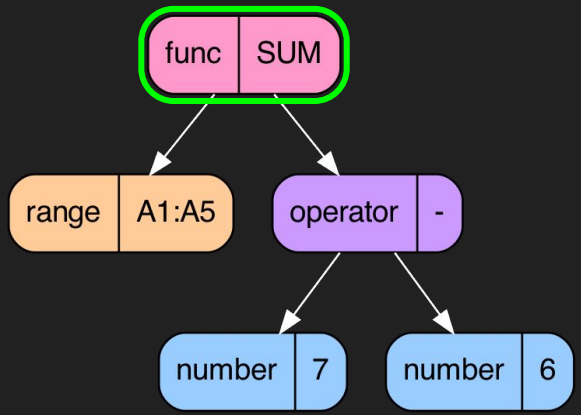
```go
// Sub calculates the - operator
func (f *functions) Sub(args ...*model.Result) *model.Result {
    if len(args) != 2 {
        return model.ErrorResult
    }

    a := args[0].ToDecimal(f.ctx)
    b := args[1].ToDecimal(f.ctx)
    return result.Decimal(a - b)
}
```



```
func calculateNode(node parser.Node, ctx *model.Context) (r *model.Result) {
   switch n := node.(type) {
...
   case *parser.Function:
       // gather the function arguments as results
       args := make([]*model.Result, len(n.Arguments))
       for i, arg := range n.Arguments {
           args[i] = calculateNode(arg, ctx)
       }

       return functions.CallByName(n.Name, ctx, args...)
```

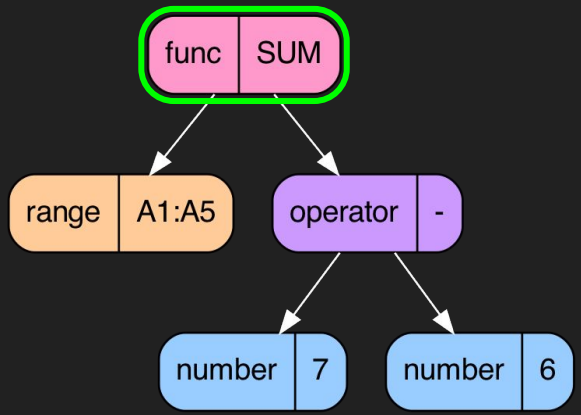


```
func calculateNode(node parser.Node, ctx *model.Context) (r *model.Result) {
   switch n := node.(type) {
...
   case *parser.Function:
       // gather the function arguments as results
       args := make([]*model.Result, len(n.Arguments))
       for i, arg := range n.Arguments {
           args[i] = calculateNode(arg, ctx)
       }

       return functions.CallByName(n.Name, ctx, args...)
```

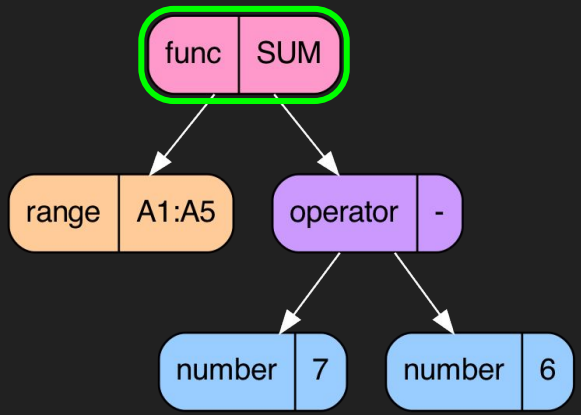


```go
func (f *functions) Sum(args ...*model.Result) *model.Result {
    var sum float64
    for _, res := range args {
        sum += res.ApplyToRange(f.ctx, f.Sum).ToDecimal(f.ctx)
    }
    return result.Decimal(sum)
}
```

```go
func (r *Result) ApplyToRange(ctx *Context, callfunc Function) *Result {
    if r.IsRange() {
        return callfunc(ctx.Cells.GetResultsInRange(&r.Ref)...)
    }
    return r
}
```



```
func calculateNode(node parser.Node, ctx *model.Context) (r *model.Result) {
   switch n := node.(type) {
...
   case *parser.Function:
       // gather the function arguments as results
       args := make([]*model.Result, len(n.Arguments))
       for i, arg := range n.Arguments {
           args[i] = calculateNode(arg, ctx)
       }

       return functions.CallByName(n.Name, ctx, args...)
```



A1		
	Text	1
	Result	1.0

B1		
	Text	=SUM(A1:A5, 7 - 6)
	Result	empty
	AST	*parser.Node

A2		
	Text	2
	Result	2.0

A3		
	Text	3
	Result	3.0

A4		
	Text	4
	Result	4.0

A5		
	Text	5
	Result	5.0

```
sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5, 7 - 6)`)

sheet.Calculate()
```

A1		
	Text	1
	Result	1.0

B1		
	Text	=SUM(A1:A5, 7 - 6)
	Result	16.0
	AST	*parser.Node

A2		
	Text	2
	Result	2.0

A3		
	Text	3
	Result	3.0

A4		
	Text	4
	Result	4.0

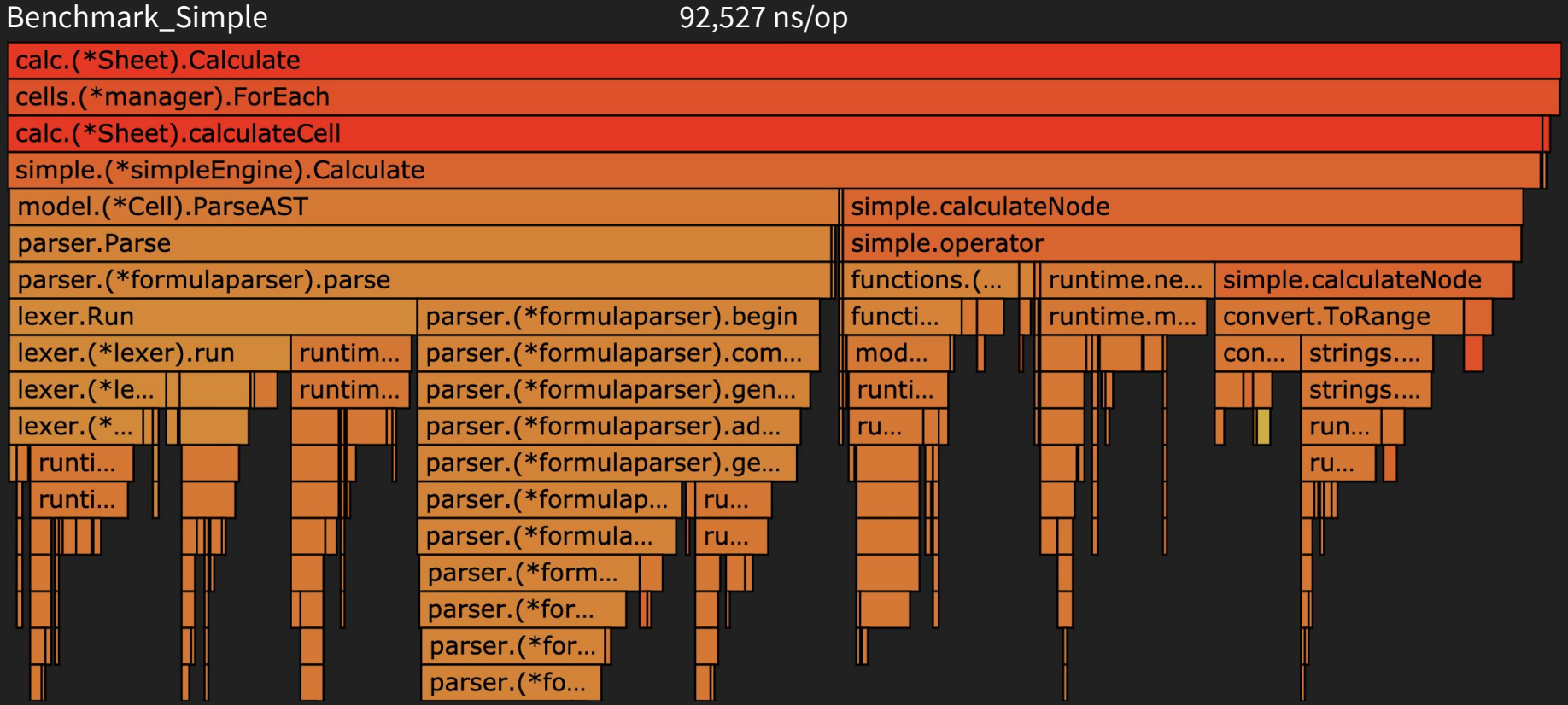
A5		
	Text	5
	Result	5.0

```
sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5,

sheet.Calculate()
```

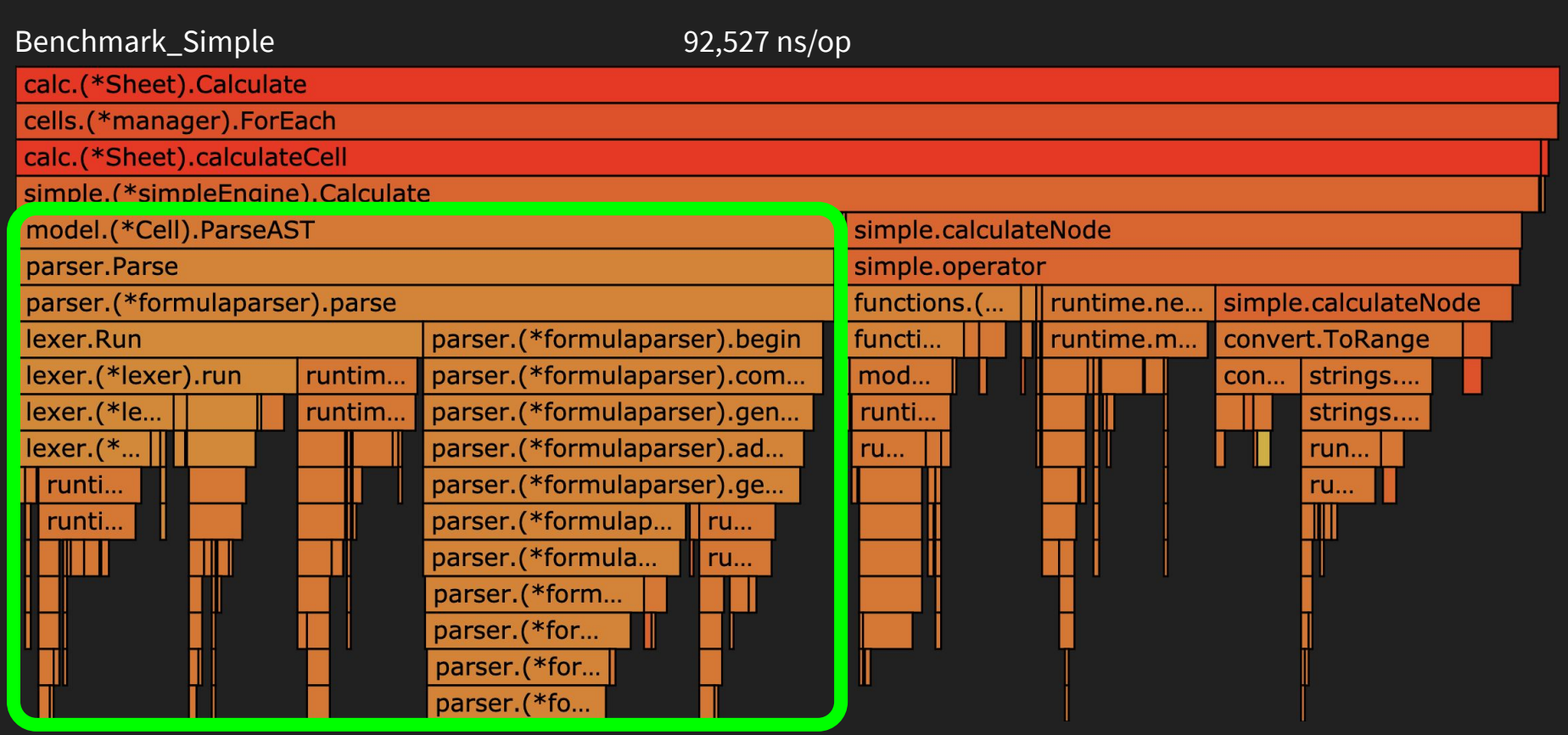
What about performance?

```
go build
./demo -engine=simple -cpuprofile=simple.cpu
go tool pprof simple.cpu
```

Calculating 100 Million Formulas



Calculating 100 Million Formulas

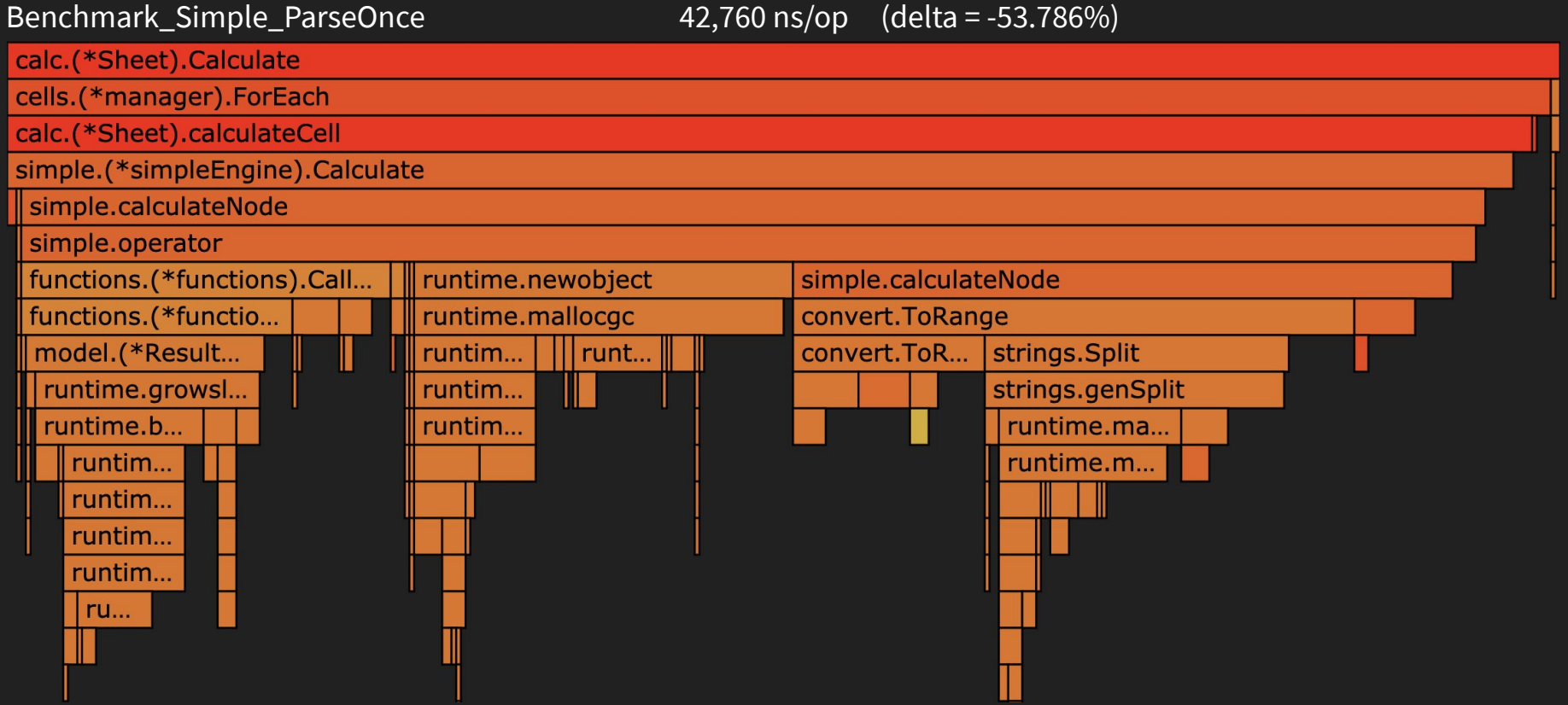


Only Parse Each Formula Once!

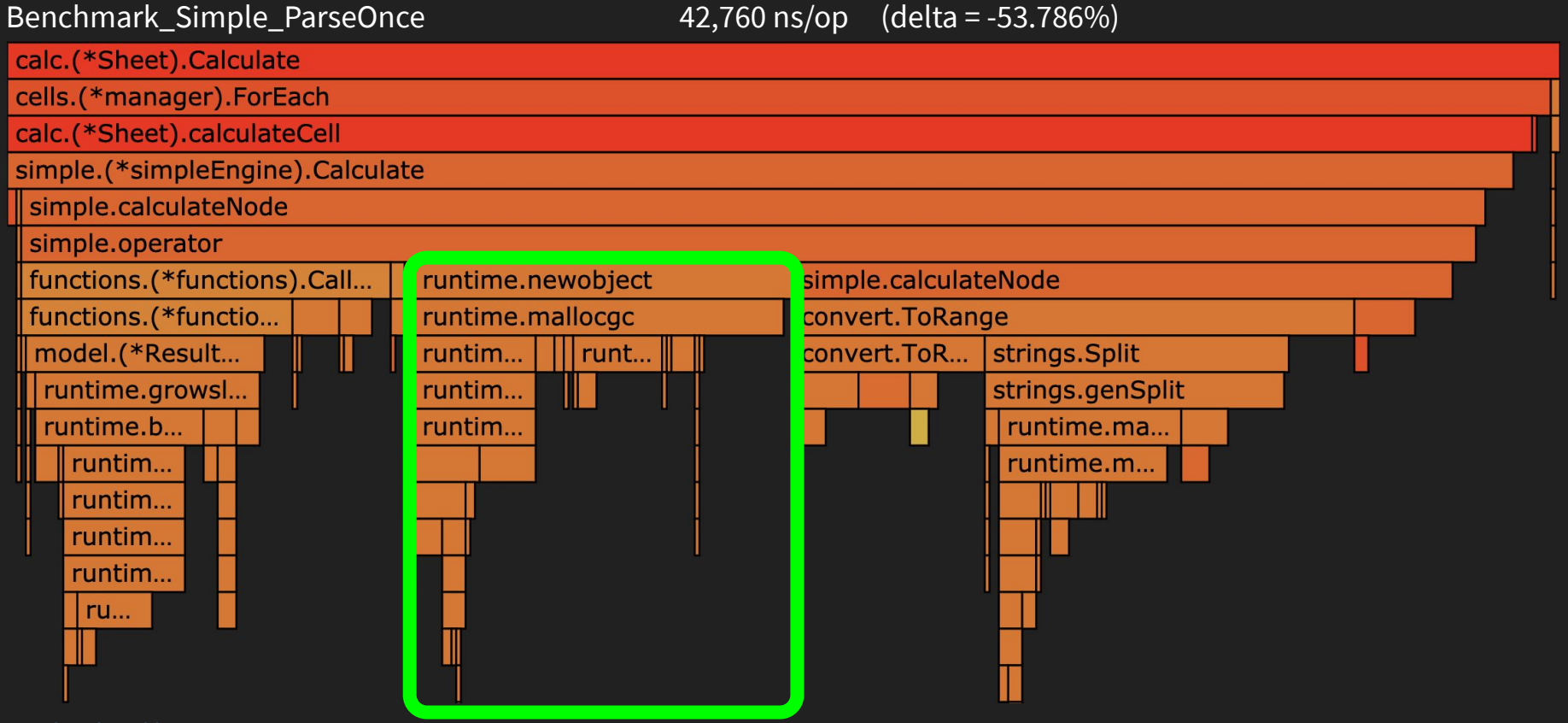
```
func (eng *simpleEngine) Parse(row, col int, ctx *model.Context) {
    cell := ctx.Cells.GetCell(row, col)
    cell.ParseAST()
}
```

```
func (eng *simpleEngine) Calculate(row, col int, ctx *model.Context) *model.Result {
    cell := ctx.Cells.GetCell(row, col)
    if !cell.IsFormula {
        return cell.Result
    }
    if cell.Result.Type != model.ResultType_Empty {
        return cell.Result
    }

    cell.ParseAST()
    result := calculateNode(cell.AST, ctx)
    cell.SetResult(result)
    return result
}
```

Only Parse Each Formula Once!



Only Parse Each Formula Once!

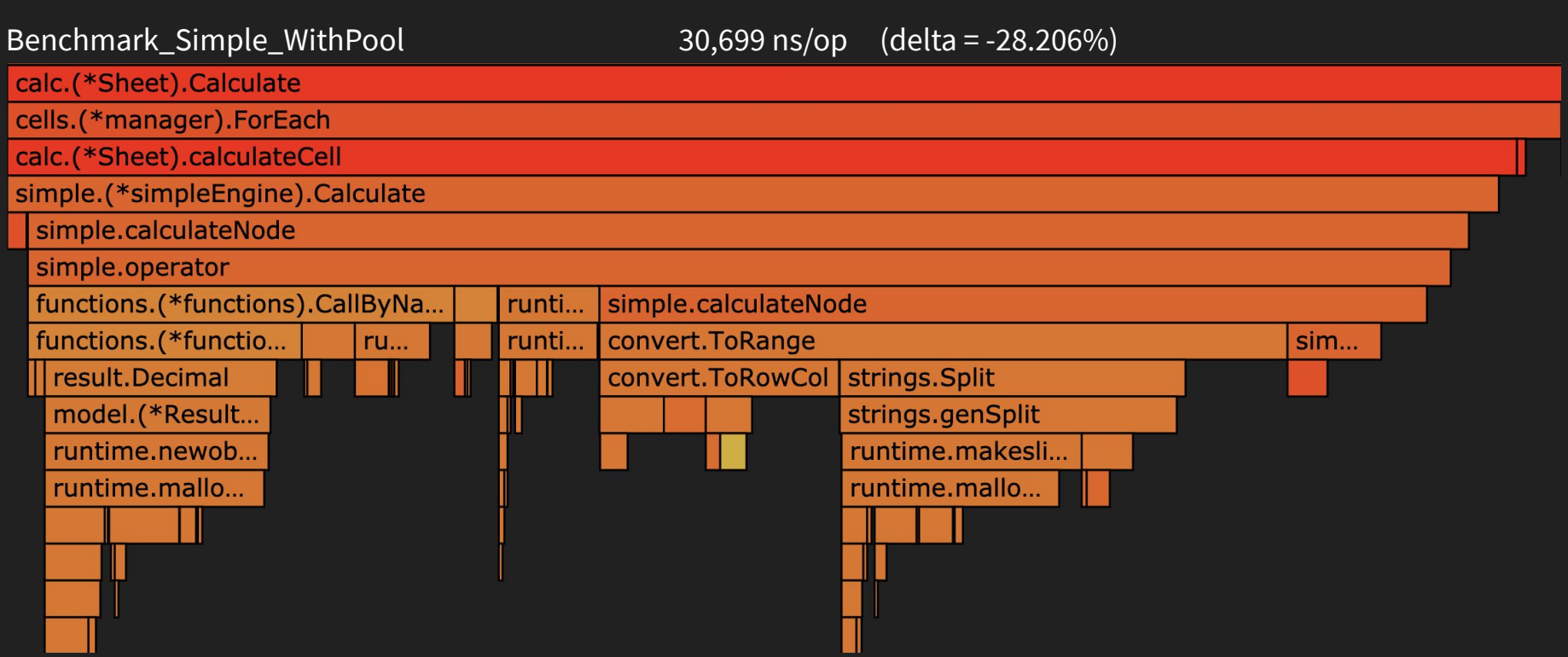


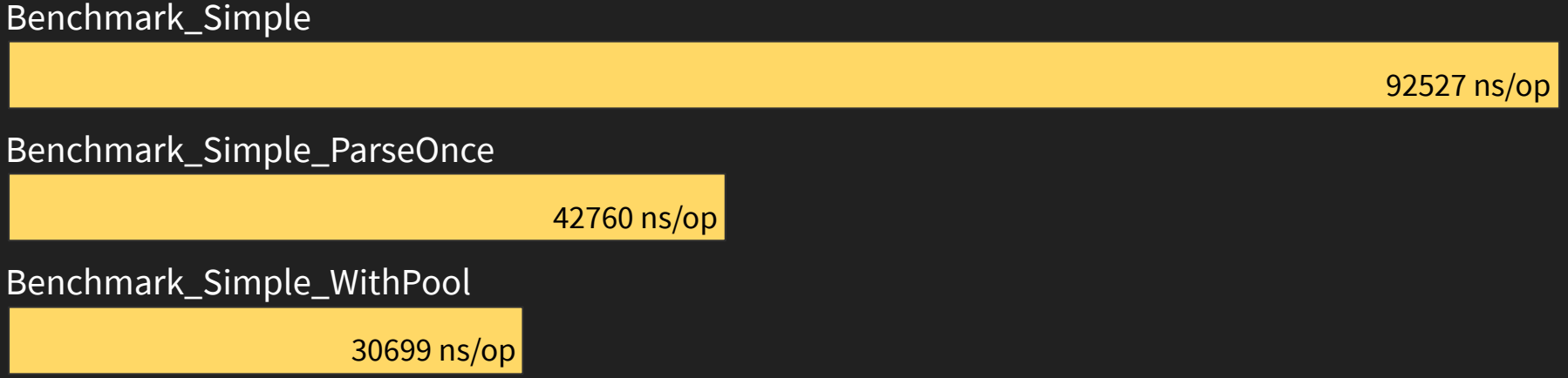
Pre-Allocate and Pool Result Objects

```go
func Decimal(val float64) *model.Result {
    return &model.Result{
        Dec:  val,
        Type: model.ResultType_Decimal,
    }
}
```

```go
func Decimal(val float64) *model.Result {
    r := model.ResultPool.Get()
    r.Type = model.ResultType_Decimal
    r.Dec = val
    return r
}
```

Pre-Allocate and Pool Result Objects




Benchmark_Simple
92527 ns/op
Benchmark_Simple_ParseOnce
42760 ns/op
Benchmark_Simple_WithPool
30699 ns/op

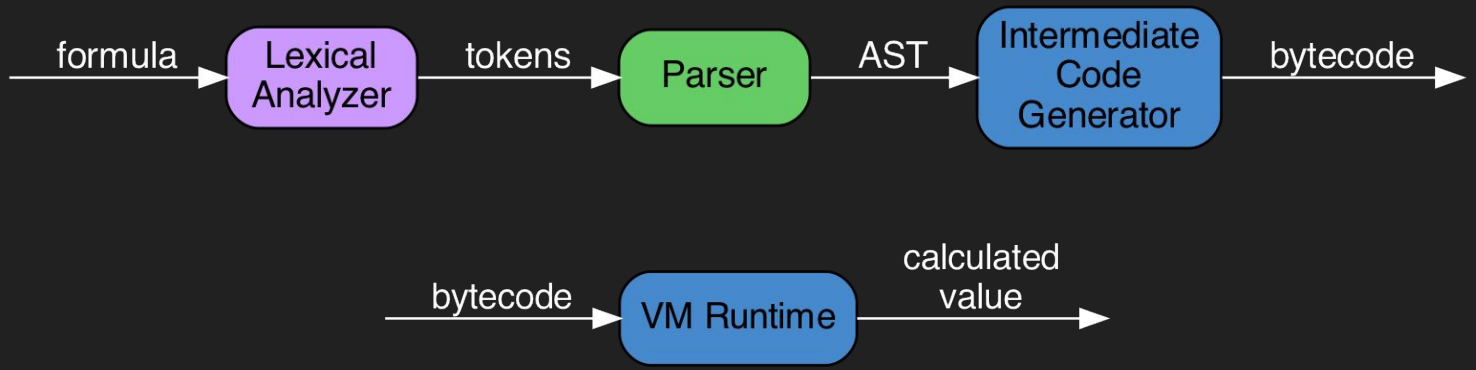
-66.82% of our original calculation time!

-66.82% of our original calculation time!

… but can we do any better?

How can we avoid reparsing formulas when reloading our spreadsheet?

Design and Build a Virtual Machine



Designing a Computer

- A virtual machine is an emulator for a computer

Designing a Computer

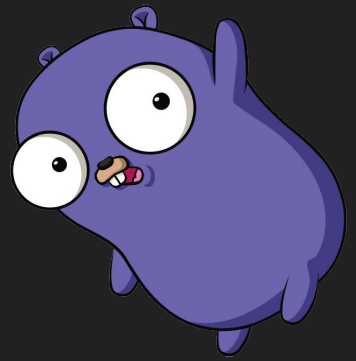
- A virtual machine is an emulator for a computer
- An intermediate language provides abstraction and portability

Designing a Computer

- A virtual machine is an emulator for a computer
- An intermediate language provides abstraction and portability
- Code executes in a domain-specific runtime environment

Designing a Computer

- A virtual machine is an emulator for a computer
- An intermediate language provides abstraction and portability
- Code executes in a domain-specific runtime environment
- The instruction set of the target runtime is called bytecode



We want a SIMPLE instruction set and a SIMPLE runtime...

A Simple Instruction Set

Literal Values

- Boolean
- Number
- String
- Cell
- Range

A Simple Instruction Set

Literal Values

- Boolean
- Number
- String
- Cell
- Range

Math Operators

- Add
- Subtract
- Multiply
- Divide

A Simple Instruction Set

Literal Values

- Boolean
- Number
- String
- Cell
- Range

Math Operators

- Add
- Subtract
- Multiply
- Divide

Functions

- SUM
- LEN
- CONCATENATE
- VLOOKUP
- …

all the other common spreadsheet functions

A Simple Instruction Set

Literal Values

- Boolean
- Number
- String
- Cell
- Range

Math Operators

- Add
- Subtract
- Multiply
- Divide

Functions

- SUM
- LEN
- CONCATENATE
- VLOOKUP
- …

all the other common spreadsheet functions

Conditionals

- IF

A Simple Instruction Set

load/push

Literal Values

- Boolean
- Number
- String
- Cell
- Range

function call

Math Operators

- Add
- Subtract
- Multiply
- Divide

Functions

- SUM
- LEN
- CONCATENATE
- VLOOKUP
- …

all the other common spreadsheet functions

cmp / jump

Conditionals

- IF

A Simple Runtime

- Let’s use a stack!
- All function arguments are popped from the stack
- All results are pushed back onto the stack

A Simple Runtime

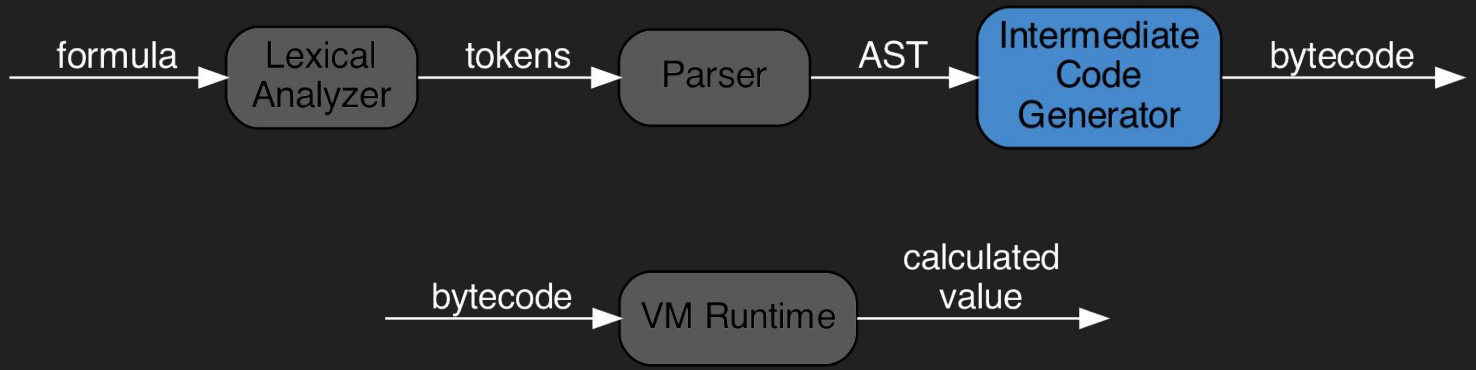
- Let’s use a stack!
- All function arguments are popped from the stack
- All results are pushed back onto the stack

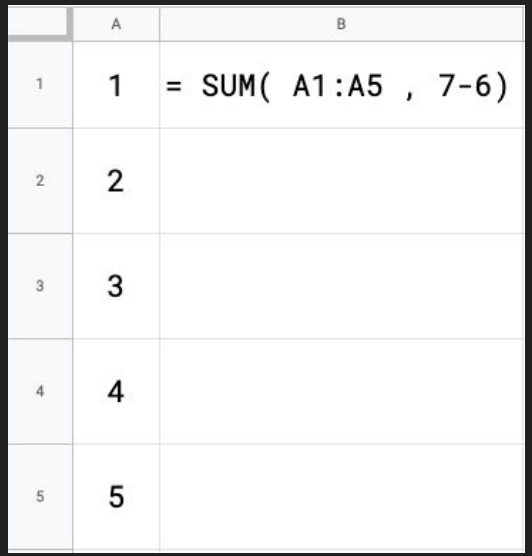
That’s it!

This is one reason why many popular VM languages use a stack-based model

So let’s learn how to generate instructions for our VM

Compiling AST to Bytecode





	A	B
1	1	= SUM(A1:A5 , 7-6)
2	2	
3	3	
4	4	
5	5	

```
engine := calc.NewEngine("vm")
sheet := calc.NewSheet(10, 10, engine)

sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5, 7 - 6)`)

sheet.Calculate()
```

```
func (*vmEngine) Parse(row, col int, ctx *model.Context) {
   cell := ctx.Cells.GetCell(row, col)
   cell.ParseAST()

   vm := New().CompileAST(cell.AST)
   cell.Bytecode = vm.Bytecode()
}
```

Warning:
Here there be dragons!

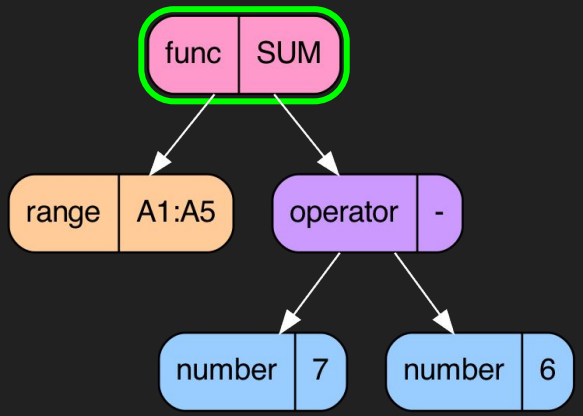
We use Postorder* Tree Traversal to compile the AST

Postorder Algorithm

1. Traverse the left subtree
2. Traverse the right subtree(s)
3. Visit the root

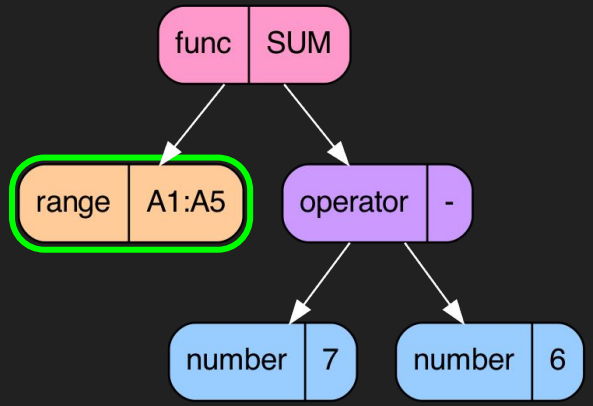
a.k.a. RPN (Reverse Polish Notation)

```
func (vm *VM) compileNode(node parser.Node) *VM {
   switch node := node.(type) {
   case *parser.Bool:
       vm.addOp(op.MakeBool(node.Value == `TRUE`))

   case *parser.Number:
       f, _ := strconv.ParseFloat(node.Value, 64)
       vm.addOp(op.MakeNumber(f))

   case *parser.String:
       vm.addOp(op.MakeString(node.Value))

   case *parser.Cell:
       vm.addOp(op.MakeRangeRef(node.Value))
   case *parser.Range:
       vm.addOp(op.MakeRangeRef(node.Value))
...

   }

   return vm
}
```



Instructions

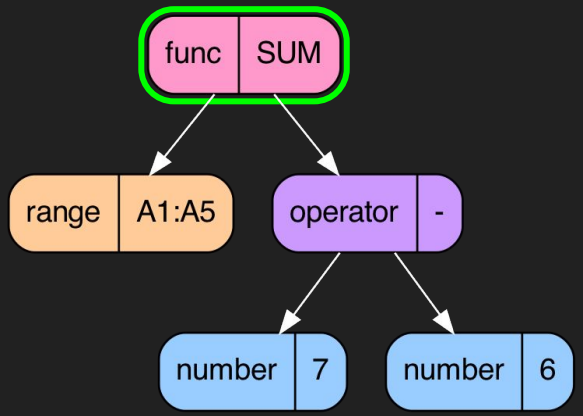
```
func (vm *VM) compileNode(node parser.Node) *VM {
   switch node := node.(type) {
...
   case *parser.Function:

       for _, arg := range node.Arguments {
           vm.compileNode(arg)
       }

       vm.addOp(op.MakeFuncCall(node.Name, len(node.Arguments)))
```



Instructions

```
range      A1:A5
```

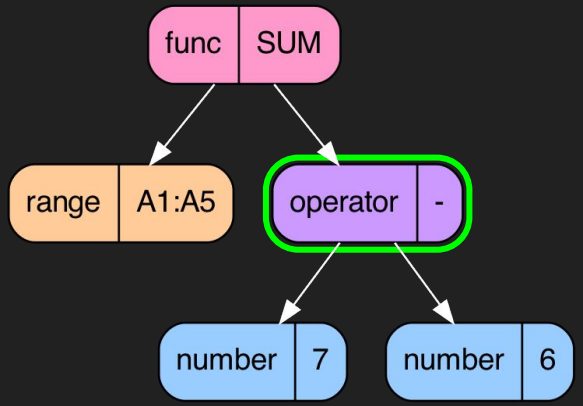
```
func (vm *VM) compileNode(node parser.Node) *VM {
    switch node := node.(type) {
...
    case *parser.Cell:
        vm.addOp(op.MakeRangeRef(node.Value))

    case *parser.Range:
        vm.addOp(op.MakeRangeRef(node.Value))
```



Instructions

```
range      A1:A5
```

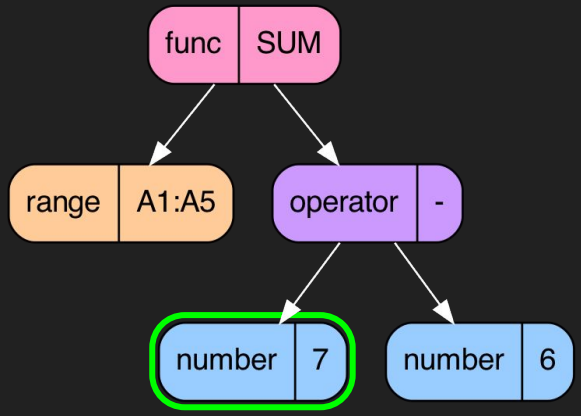
```
func (vm *VM) compileNode(node parser.Node) *VM {
    switch node := node.(type) {
...
    case *parser.Function:

        for _, arg := range node.Arguments {
            vm.compileNode(arg)
        }

        vm.addOp(op.MakeFuncCall(node.Name, len(node.Arguments)))
```



Instructions

```
range     A1:A5
```

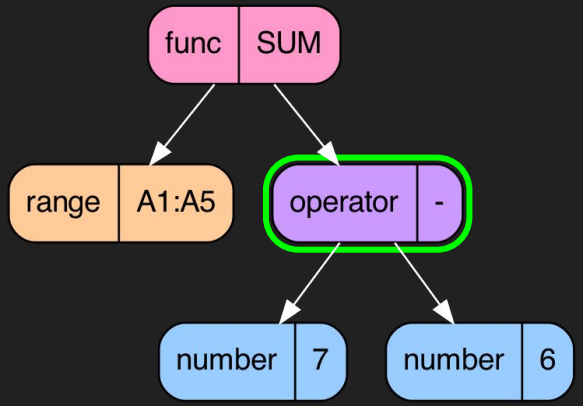
```
func (vm *VM) compileNode(node parser.Node) *VM {
   switch node := node.(type) {
...
   case *parser.Operator:

       if node.LHS != nil {
           vm.compileNode(node.LHS) // left subtree
       }

       vm.compileNode(node.RHS) // right subtree

       funcName := functions.OperatorNames(node.Operation)
       vm.addOp(op.MakeFuncCall(funcName, 2))
```



Instructions

```
range     A1:A5
uint      7
```

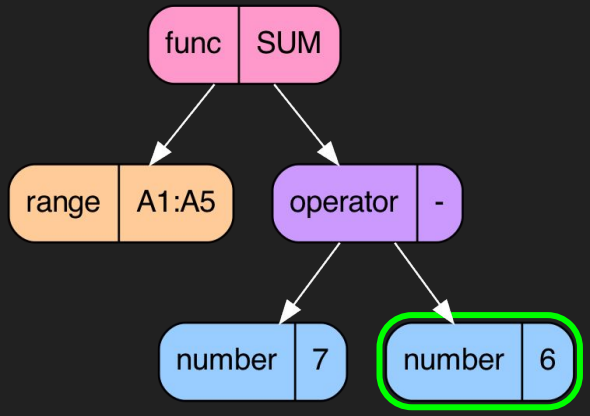
```
func (vm *VM) compileNode(node parser.Node) *VM {
   switch node := node.(type) {
...
   case *parser.Number:

       f, err := strconv.ParseFloat(node.Value, 64)
       if err != nil {
           panic(err)
       }
       vm.addOp(op.MakeNumber(f))
```



Instructions

```
range     A1:A5
uint      7
```

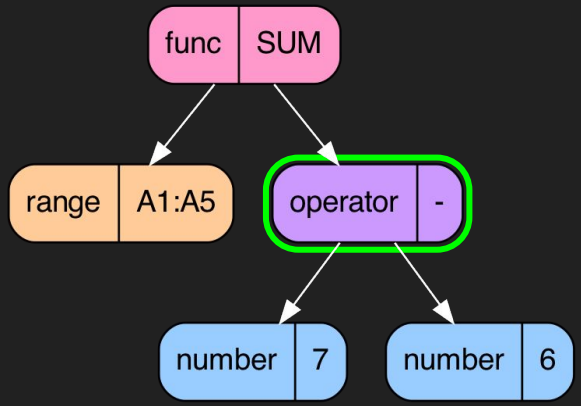
```
func (vm *VM) compileNode(node parser.Node) *VM {
   switch node := node.(type) {
...
   case *parser.Operator:

       if node.LHS != nil {
           vm.compileNode(node.LHS) // left subtree
       }

       vm.compileNode(node.RHS) // right subtree

       funcName := functions.OperatorNames(node.Operation)
       vm.addOp(op.MakeFuncCall(funcName, 2))
```



Instructions

```
range     A1:A5
uint      7
uint      6
```

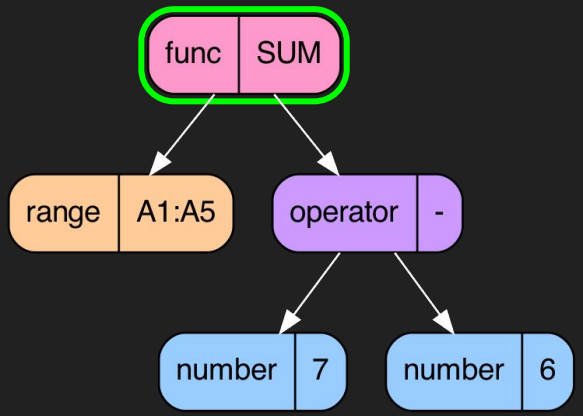
```
func (vm *VM) compileNode(node parser.Node) *VM {
   switch node := node.(type) {
...
   case *parser.Number:

       f, err := strconv.ParseFloat(node.Value, 64)
       if err != nil {
           panic(err)
       }
       vm.addOp(op.MakeNumber(f))
```



Instructions

```
range    A1:A5
uint     7
uint     6
call     SUB.2
```

```
func (vm *VM) compileNode(node parser.Node) *VM {
   switch node := node.(type) {
...
   case *parser.Operator:

       if node.LHS != nil {
           vm.compileNode(node.LHS) // left subtree
       }

       vm.compileNode(node.RHS) // right subtree

       funcName := functions.OperatorNames(node.Operation)
       vm.addOp(op.MakeFuncCall(funcName, 2))
```



Instructions

```
range    A1:A5
uint     7
uint     6
call     SUB.2
call     SUM.2
```

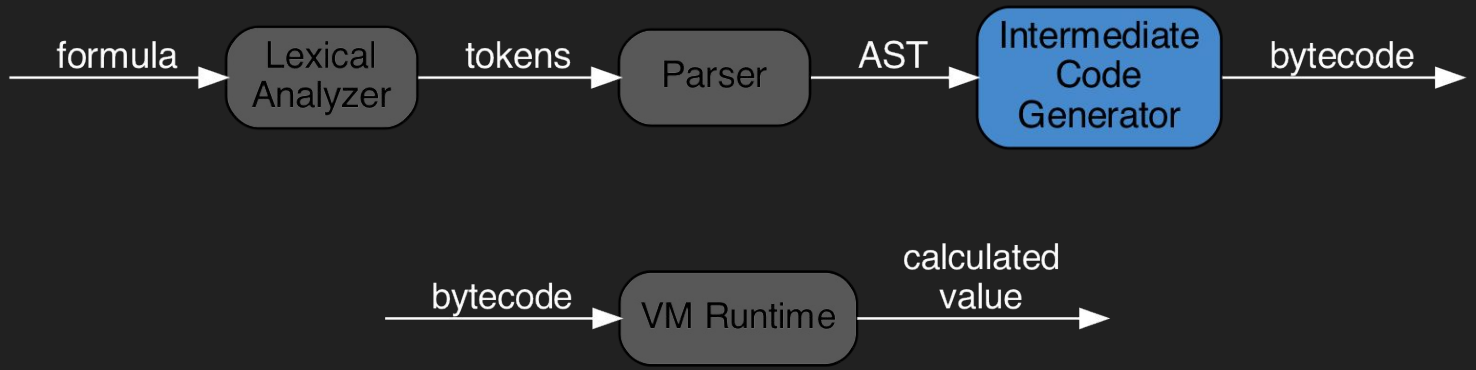
```
func (vm *VM) compileNode(node parser.Node) *VM {
   switch node := node.(type) {
...
   case *parser.Function:

       for _, arg := range node.Arguments {
           vm.compileNode(arg)
       }

       vm.addOp(op.MakeFuncCall(node.Name, len(node.Arguments)))
```

Formula: `= SUM ( A1:A5 , 7 - 6 )`

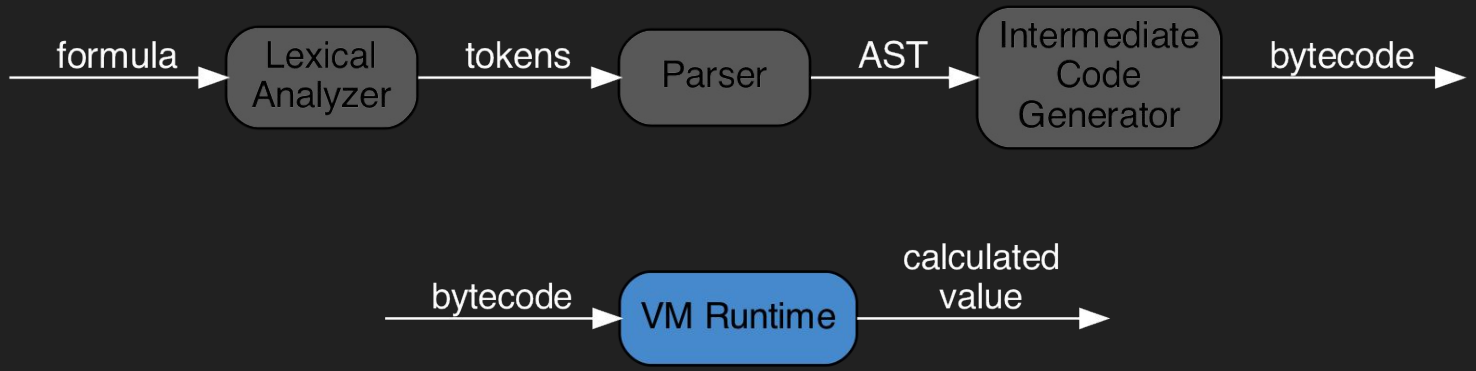
```
1d 00 00 00 08 00 10 07 10 06 22 02 22 05
```

```
0x0000:    1d 00 00 00 08 00    range    A1:A5
0x0006:    10 07                uint     7
0x0008:    10 06                uint     6
0x000a:    22 02                call     SUB.2
0x000c:    22 05                call     SUM.2
0x000e:                         return
```

We have bytecode, now what?



Building the VM Runtime



A1	Text	1
	Result	1.0

B1	Text	=SUM(A1:A5, 7 - 6)
	Result	empty
	AST	*parser.Node

A2	Text	2
	Result	2.0

A3	Text	3
	Result	3.0

A4	Text	4
	Result	4.0

A5	Text	5
	Result	5.0

```
sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5, 7 - 6)`)

sheet.Calculate()
```

```go
func (eng *vmEngine) Calculate(row, col int, ctx *model.Context) (result *model.Result) {
    cell := ctx.Cells.GetCell(row, col)
    if !cell.IsFormula {
        return cell.Result
    }
    if cell.Result.Type != model.ResultType_Empty {
        return cell.Result
    }

    vm := New(cell.Bytecode...)
    result = vm.Run(ctx)
    cell.SetResult(result)
    return result
}
```

```
func (vm *VM) Run(ctx *model.Context) *model.Result {
   vm.ip = 0
   for vm.ip < len(vm.bytecode) {
       code := vm.nextOpcode()
       switch code {
...
...
       default:
           continue
       }
   }

   result := vm.stack.Pop()
   return result
}
```

Instructions

```
range    A1:A5
uint     7
uint     6
call     SUB.2
call     SUM.2
```

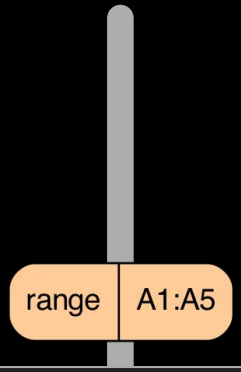
VM Stack

```go
func (vm *VM) Run(ctx *model.Context) *model.Result {
...
    code := vm.nextOpcode()
    switch code {

    case op.Opcode_RangeRef:
        rng := vm.readRange()

        // let's calculate ALL the cells in this range!
        ctx.Cells.ForEach(&rng, func(cell *model.Cell) {
            res := ctx.Engine.Calculate(cell.Row, cell.Col, ctx)
        })

        vm.stack.Push(result.Range(rng))
```

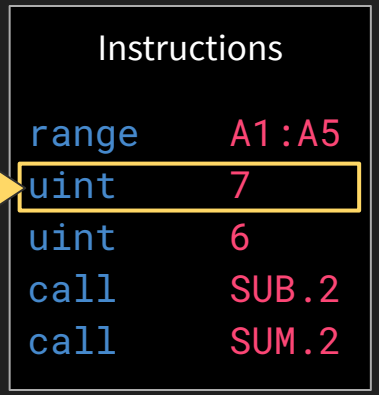
Instructions

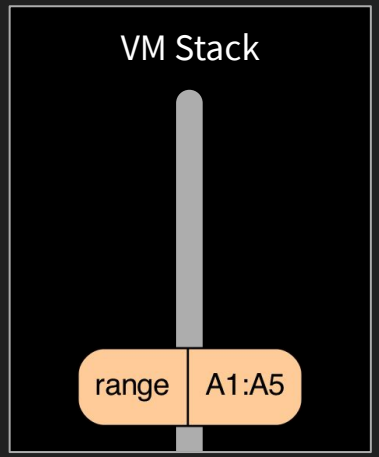
```
range    A1:A5
uint     7
uint     6
call     SUB.2
call     SUM.2
```

VM Stack

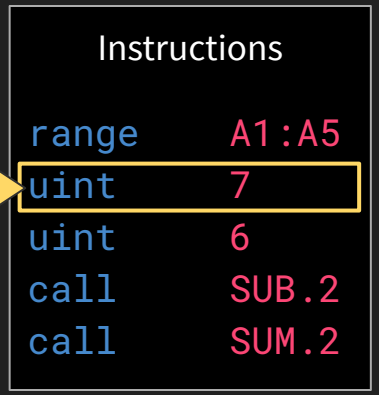



```
func (vm *VM) Run(ctx *model.Context) *model.Result {
...
    code := vm.nextOpcode()
    switch code {

    case op.Opcode_RangeRef:
        rng := vm.readRange()

        // let's calculate ALL the cells in this range!
        ctx.Cells.ForEach(&rng, func(cell *model.Cell) {
            res := ctx.Engine.Calculate(cell.Row, cell.Col, ctx)
        })

        vm.stack.Push(result.Range(rng))
```

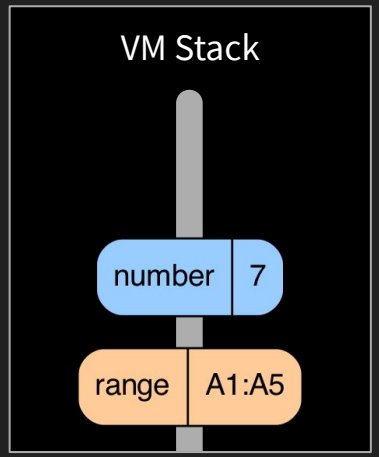


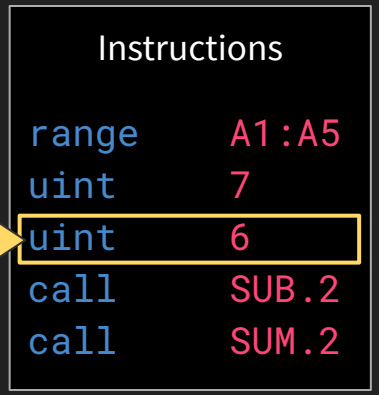


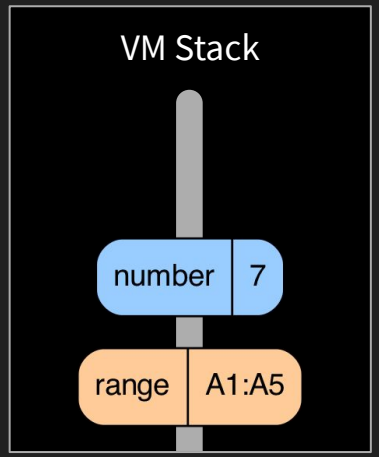

```
func (vm *VM) Run(ctx *model.Context) *model.Result {
...
    code := vm.nextOpcode()
    switch code {

    case op.Opcode_Uint:
        vm.stack.Push(result.Decimal(float64(vm.readVarUint())))
```




```
func (vm *VM) Run(ctx *model.Context) *model.Result {
...
    code := vm.nextOpcode()
    switch code {

    case op.Opcode_Uint:
        vm.stack.Push(result.Decimal(float64(vm.readVarUint())))
```







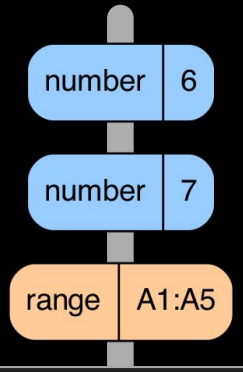

```
func (vm *VM) Run(ctx *model.Context) *model.Result {
...
    code := vm.nextOpcode()
    switch code {

    case op.Opcode_Uint:
        vm.stack.Push(result.Decimal(float64(vm.readVarUint())))
```

Instructions

```
range    A1:A5
uint     7
uint     6
call     SUB.2
call     SUM.2
```

```
func (vm *VM) Run(ctx *model.Context) *model.Result {
...
    code := vm.nextOpcode()
    switch code {

    case op.Opcode_Uint:
        vm.stack.Push(result.Decimal(float64(vm.readVarUint())))
```

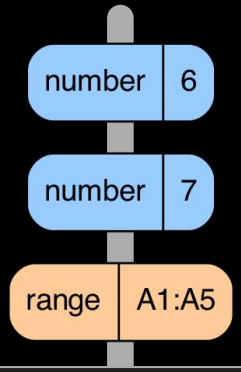
VM Stack



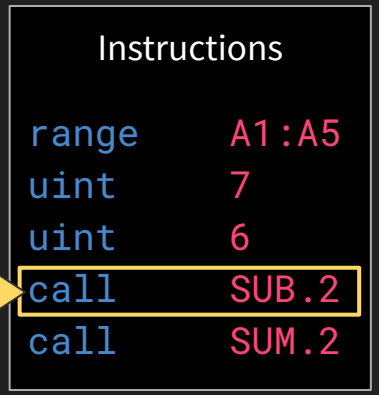
Instructions

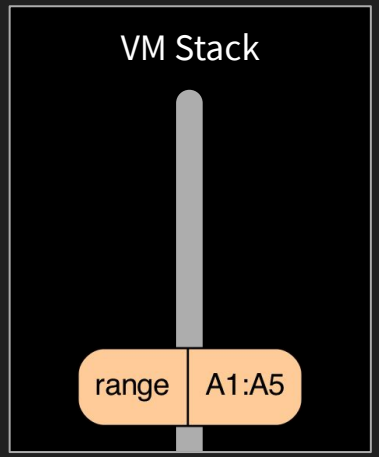
```
range     A1:A5
uint      7
uint      6
call      SUB.2
call      SUM.2
```

VM Stack

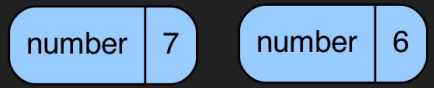



```
func (vm *VM) Run(ctx *model.Context) *model.Result {
...
    code := vm.nextOpcode()
    switch code {

    case op.Opcode_Call2:
        funcNum := int(vm.readVarUint())
        numArgs := 2
        args := vm.stack.PopN(numArgs)
        result := ctx.Functions.CallByID(funcNum, args...)
        vm.stack.Push(result)
```





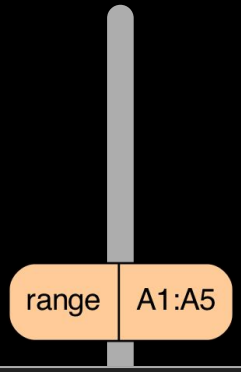

```
func (vm *VM) Run(ctx *model.Context) *model.Result {
...
    code := vm.nextOpcode()
    switch code {

    case op.Opcode_Call2:
        funcNum := int(vm.readVarUint())
        numArgs := 2
        args := vm.stack.PopN(numArgs)
        result := ctx.Functions.CallByID(funcNum, args...)
        vm.stack.Push(result)
```



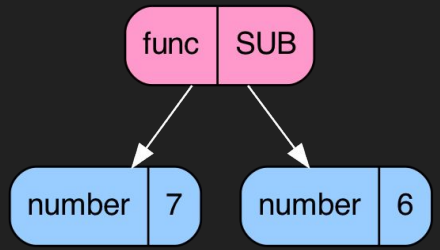
Instructions

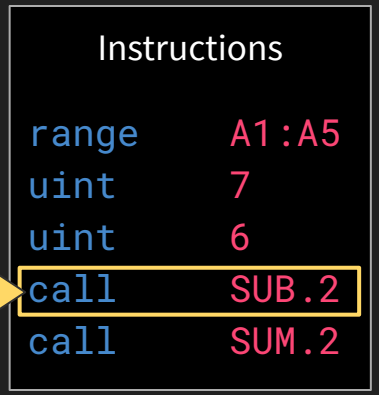
```
range    A1:A5
uint     7
uint     6
call     SUB.2
call     SUM.2
```

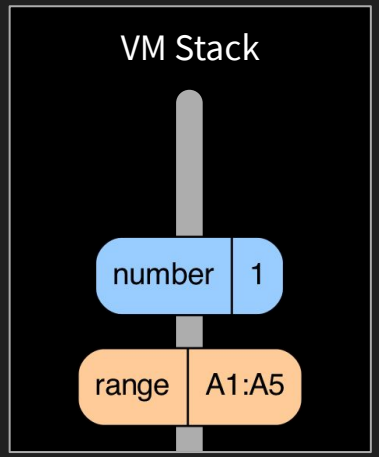
VM Stack



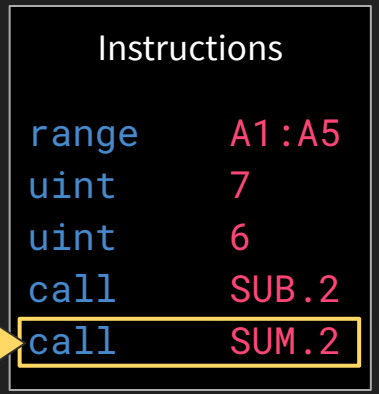

```
func (vm *VM) Run(ctx *model.Context) *model.Result {
...
    code := vm.nextOpcode()
    switch code {

    case op.Opcode_Call2:
        funcNum := int(vm.readVarUint())
        numArgs := 2
        args := vm.stack.PopN(numArgs)
        result := ctx.Functions.CallByID(funcNum, args...)
        vm.stack.Push(result)
```

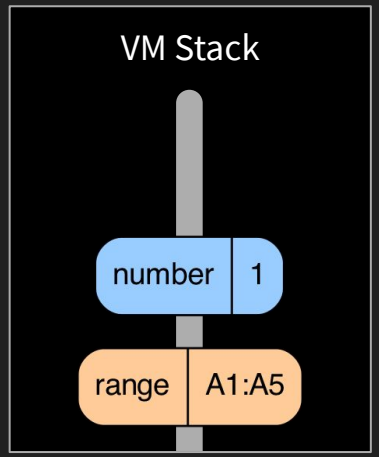




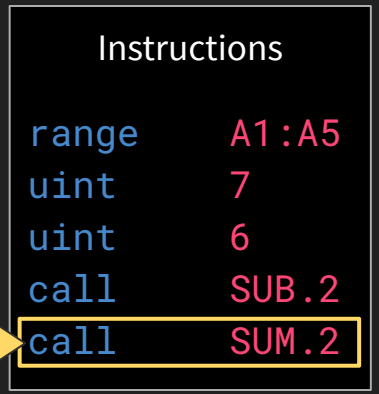


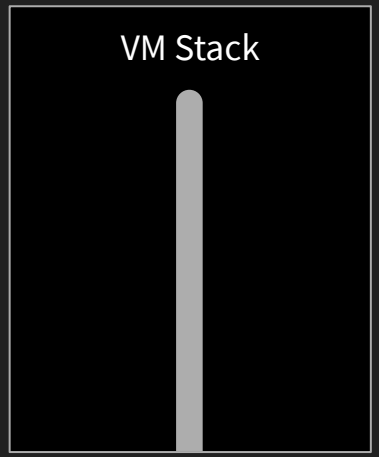

```
func (vm *VM) Run(ctx *model.Context) *model.Result {
...
    code := vm.nextOpcode()
    switch code {

    case op.Opcode_Call2:
        funcNum := int(vm.readVarUint())
        numArgs := 2
        args := vm.stack.PopN(numArgs)
        result := ctx.Functions.CallByID(funcNum, args...)
        vm.stack.Push(result)
```



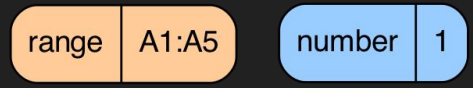



```
func (vm *VM) Run(ctx *model.Context) *model.Result {
...
    code := vm.nextOpcode()
    switch code {

    case op.Opcode_Call2:
        funcNum := int(vm.readVarUint())
        numArgs := 2
        args := vm.stack.PopN(numArgs)
        result := ctx.Functions.CallByID(funcNum, args...)
        vm.stack.Push(result)
```





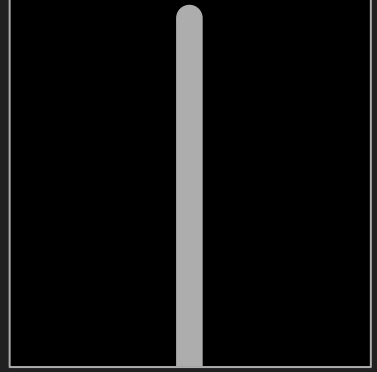

```
func (vm *VM) Run(ctx *model.Context) *model.Result {
...
    code := vm.nextOpcode()
    switch code {

    case op.Opcode_Call2:
        funcNum := int(vm.readVarUint())
        numArgs := 2
        args := vm.stack.PopN(numArgs)
        result := ctx.Functions.CallByID(funcNum, args...)
        vm.stack.Push(result)
```



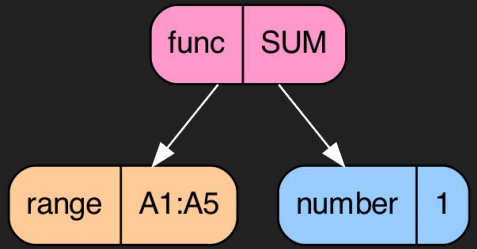
Instructions

```
range    A1:A5
uint     7
uint     6
call     SUB.2
call     SUM.2
```

VM Stack



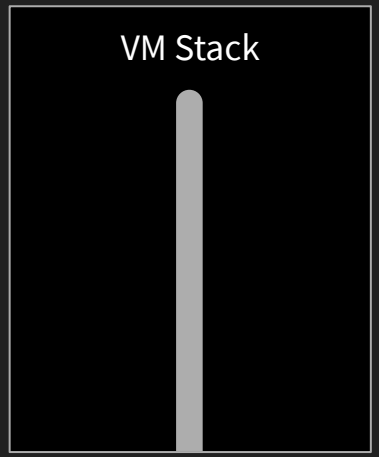
```
func (vm *VM) Run(ctx *model.Context) *model.Result {
...
    code := vm.nextOpcode()
    switch code {

    case op.Opcode_Call2:
        funcNum := int(vm.readVarUint())
        numArgs := 2
        args := vm.stack.PopN(numArgs)
        result := ctx.Functions.CallByID(funcNum, args...)
        vm.stack.Push(result)
```



Instructions

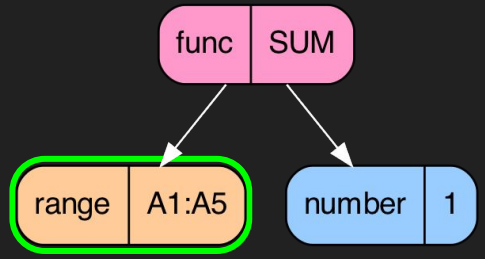
```
range    A1:A5
uint     7
uint     6
call     SUB.2
call     SUM.2
```

VM Stack



```go
func (f *functions) Sum(args ...*model.Result) *model.Result {
    var sum float64
    for _, res := range args {
        sum += res.ApplyToRange(f.ctx, f.Sum).ToDecimal(f.ctx)
    }
    return result.Decimal(sum)
}
```

```go
func (r *Result) ApplyToRange(ctx *Context, callfunc Function) *Result {
    if r.IsRange() {
        return callfunc(ctx.Cells.GetResultsInRange(&r.Ref)...)
    }
    return r
}
```



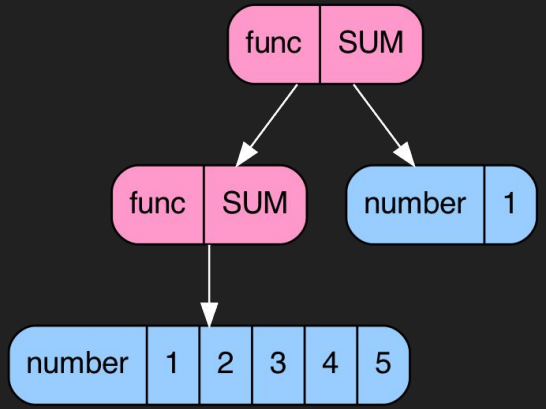
Instructions

```
range    A1:A5
uint     7
uint     6
call     SUB.2
call     SUM.2
```

VM Stack

```
func (f *functions) Sum(args ...*model.Result) *model.Result {
    var sum float64
    for _, res := range args {
        sum += res.ApplyToRange(f.ctx, f.Sum).ToDecimal(f.ctx)
    }
    return result.Decimal(sum)
}
```

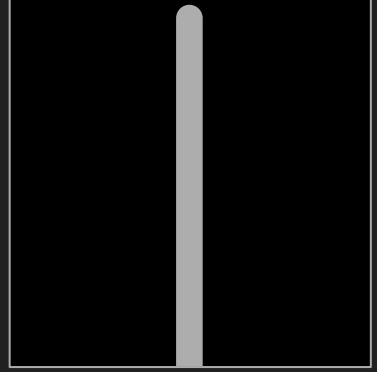
```
func (r *Result) ApplyToRange(ctx *Context, callfunc Function) *Result {
    if r.IsRange() {
        return callfunc(ctx.Cells.GetResultsInRange(&r.Ref)...)
    }
    return r
}
```



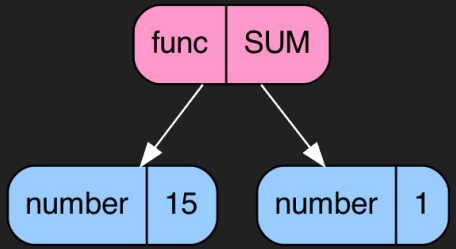
Instructions

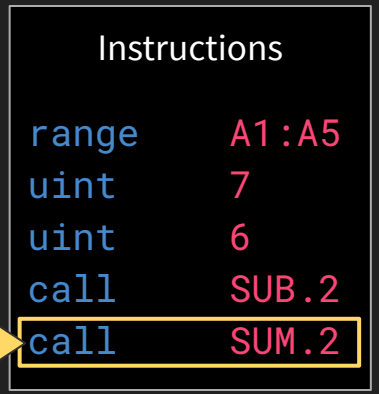
```
range    A1:A5
uint     7
uint     6
call     SUB.2
call     SUM.2
```

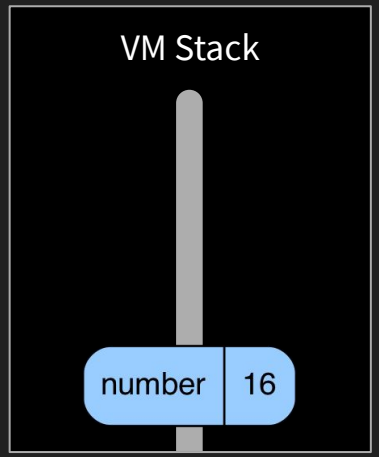
VM Stack



```
func (vm *VM) Run(ctx *model.Context) *model.Result {
...
    code := vm.nextOpcode()
    switch code {

    case op.Opcode_Call2:
        funcNum := int(vm.readVarUint())
        numArgs := 2
        args := vm.stack.PopN(numArgs)
        result := ctx.Functions.CallByID(funcNum, args...)
        vm.stack.Push(result)
```








```
func (vm *VM) Run(ctx *model.Context) *model.Result {
...
    code := vm.nextOpcode()
    switch code {

    case op.Opcode_Call2:
        funcNum := int(vm.readVarUint())
        numArgs := 2
        args := vm.stack.PopN(numArgs)
        result := ctx.Functions.CallByID(funcNum, args...)
        vm.stack.Push(result)
```

A1		
	Text	1
	Result	1.0

B1		
	Text	=SUM(A1:A5, 7 - 6)
	Result	empty
	AST	*parser.Node

A2		
	Text	2
	Result	2.0

A3		
	Text	3
	Result	3.0

A4		
	Text	4
	Result	4.0

A5		
	Text	5
	Result	5.0

```
sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5, 7 - 6)`)

sheet.Calculate()
```

A1		
	Text	1
	Result	1.0

B1		
	Text	=SUM(A1:A5, 7 - 6)
	Result	16.0
	AST	*parser.Node

A2		
	Text	2
	Result	2.0

A3		
	Text	3
	Result	3.0

A4		
	Text	4
	Result	4.0

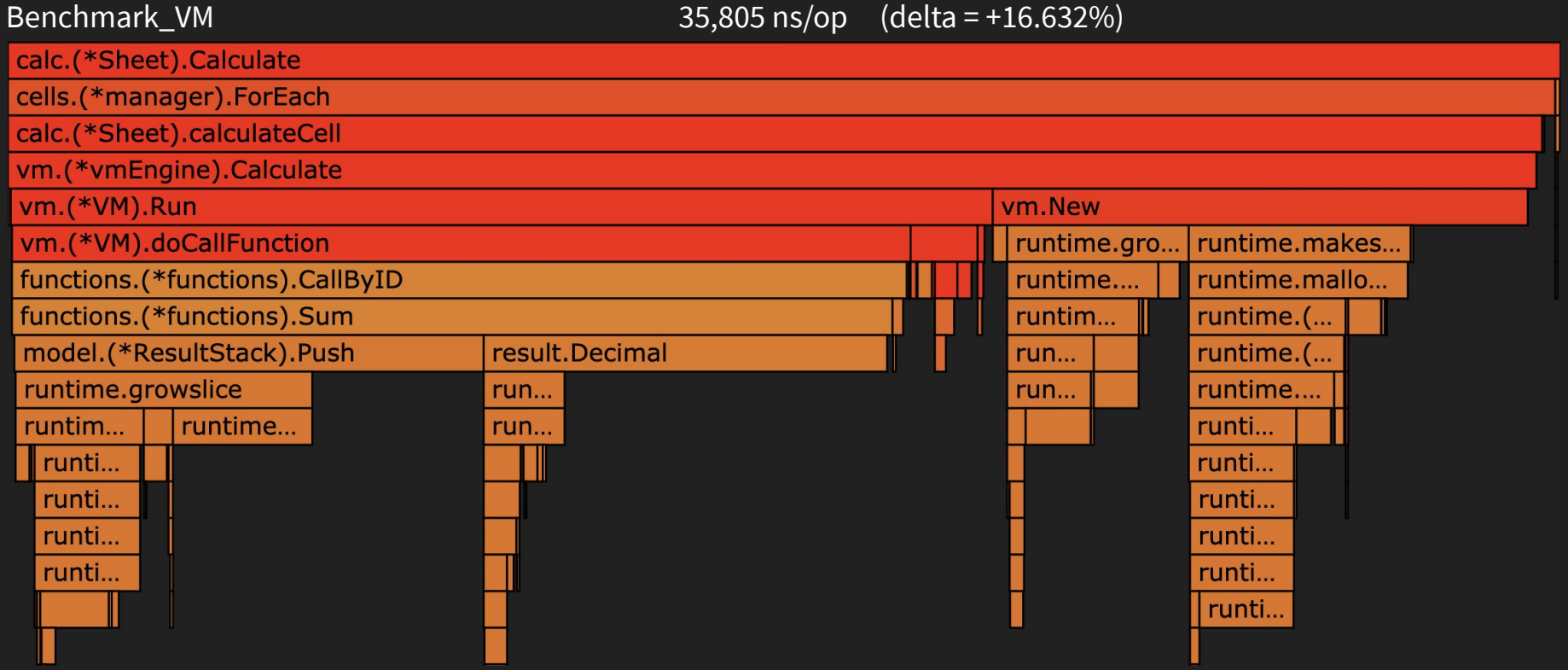
A5		
	Text	5
	Result	5.0

```
sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5,

sheet.Calculate()
```

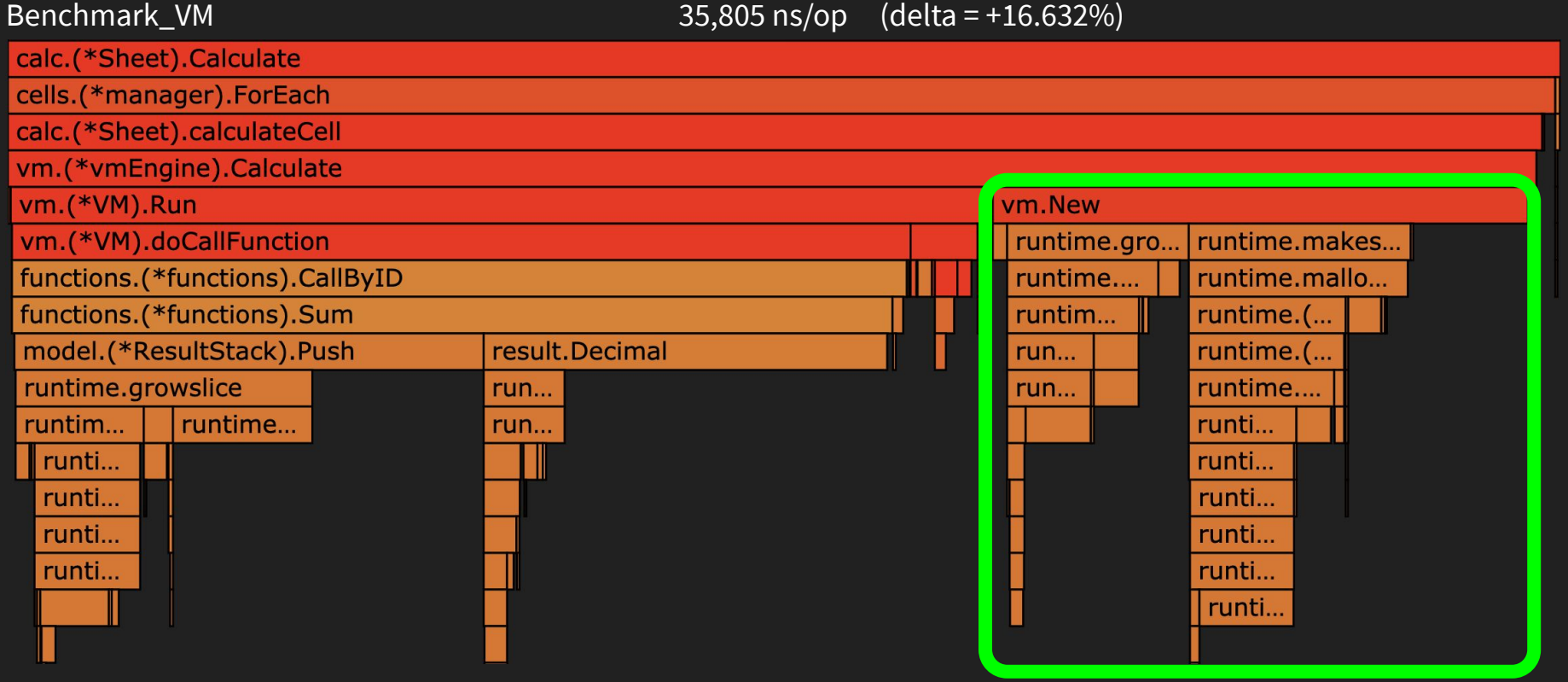
What about performance?

```
go build
./demo -engine=vm -cpuprofile=vm.cpu
go tool pprof vm.cpu
```

Calculating with our Virtual Machine



Calculating with our Virtual Machine

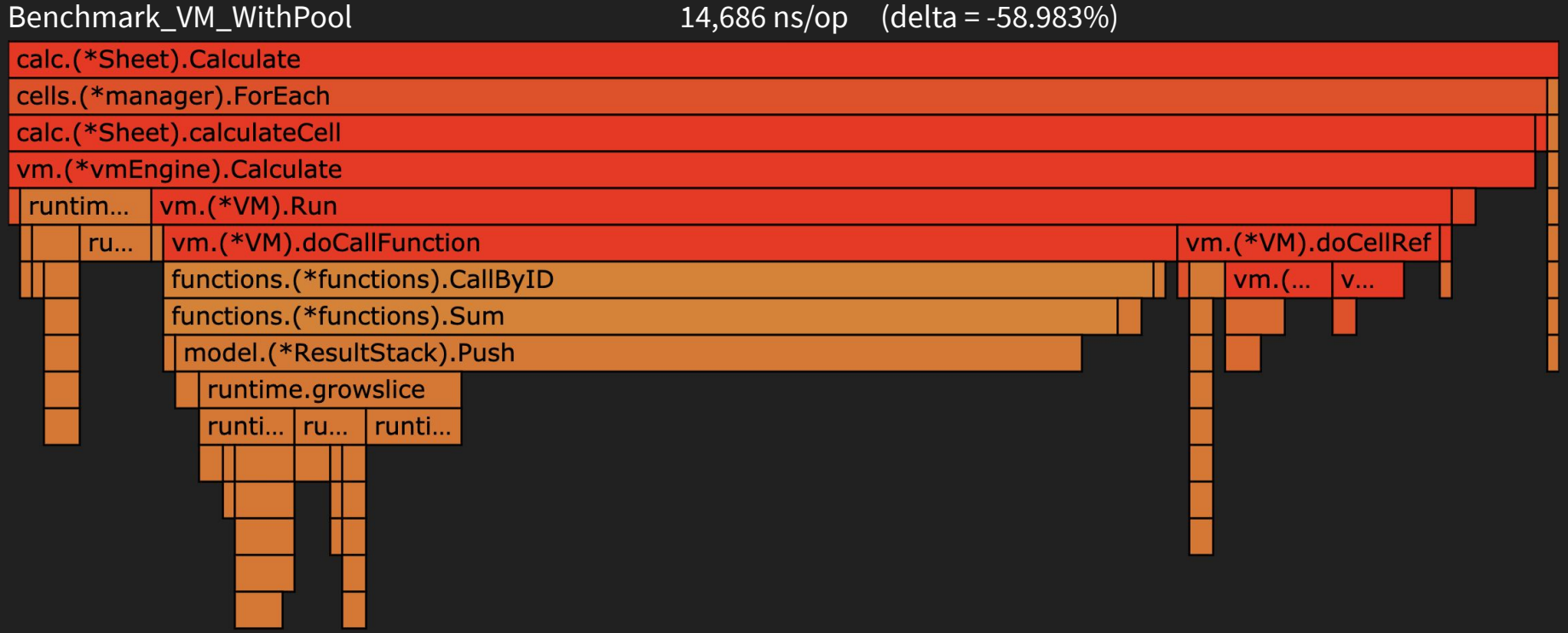


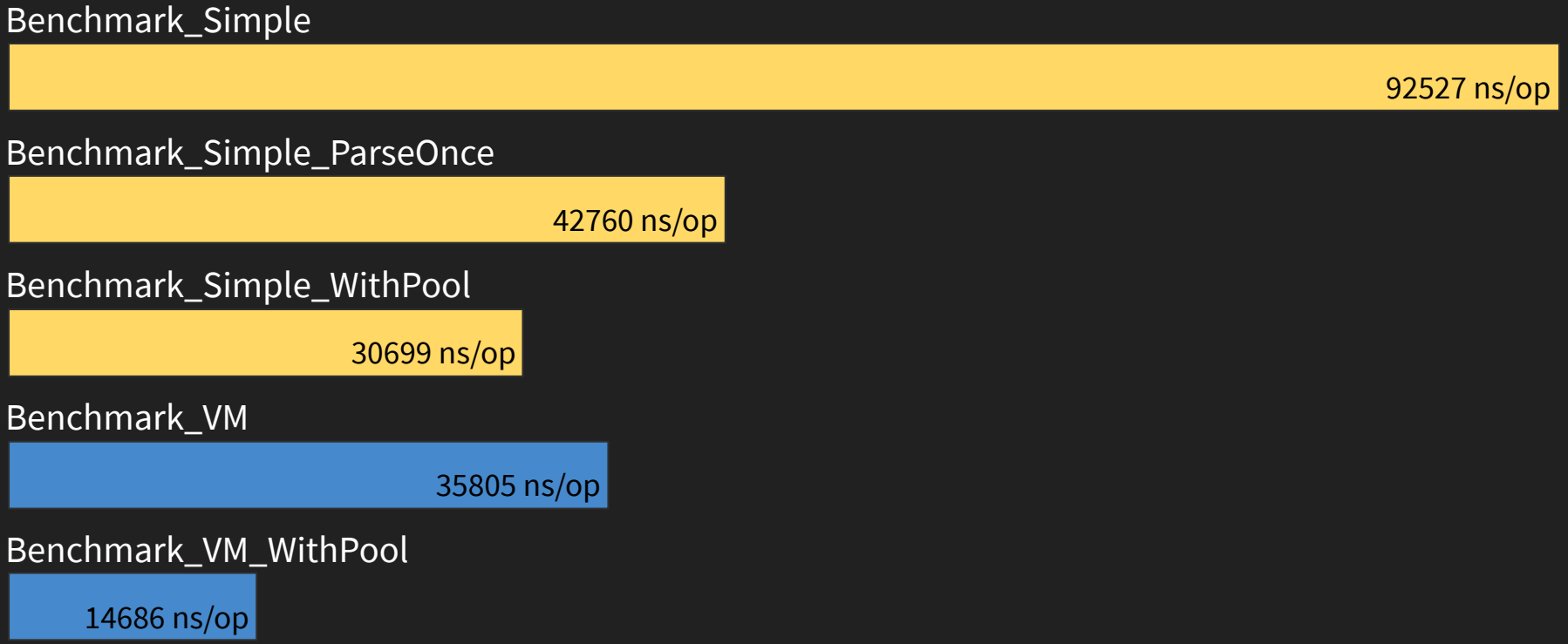
Reusable Pool of Virtual Machines

```
func (eng *vmEngine) Calculate(...) {
   vm := New(cell.Bytecode...)
   result = vm.Run(ctx)

   cell.SetResult(result)
   return result
}
```

```
func (eng *vmEngine) Calculate(...) {
   vm := NewFromPool(cell.Bytecode...)
   result = vm.Run(ctx)

   vm.Reset()
   Pool = append(Pool, vm)

   cell.SetResult(result)
   return result
}
```

Reusable Pool of Virtual Machines




Benchmark_Simple
92527 ns/op
Benchmark_Simple_ParseOnce
42760 ns/op
Benchmark_Simple_WithPool
30699 ns/op
Benchmark_VM
35805 ns/op
Benchmark_VM_WithPool
14686 ns/op

-84.13% of our original calculation time!

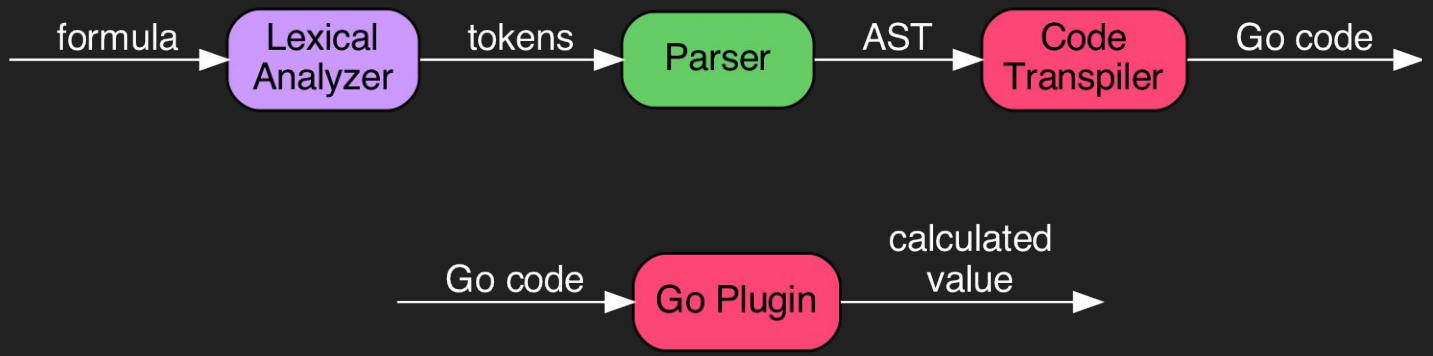
-84.13% of our original calculation time!

… but we're still not done yet!

Our VM compiler works pretty well, but we’re not experts at this.

What if we could use the Go Compiler to compile our formulas?

Compiling Go Plugins at Runtime



Go Plugins

- The Go plugin system compiles packages as shared object libraries

Go Plugins

- The Go plugin system compiles packages as shared object libraries
- A plugin is a Go main package with exported functions and variables

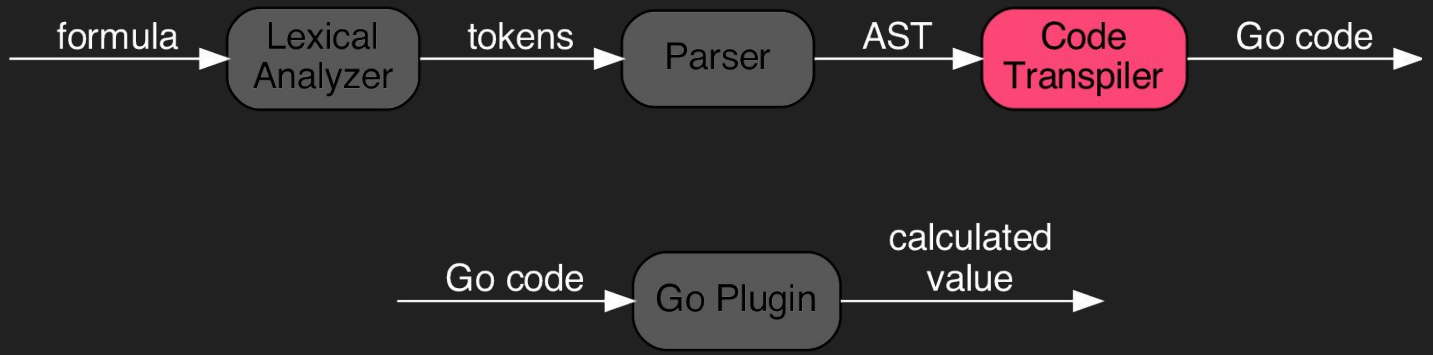
Go Plugins

- The Go plugin system compiles packages as shared object libraries
- A plugin is a Go main package with exported functions and variables
- Compiled libraries can be loaded dynamically at runtime

Go Plugins

- The Go plugin system compiles packages as shared object libraries
- A plugin is a Go main package with exported functions and variables
- Compiled libraries can be loaded dynamically at runtime
- Only supported on Linux and MacOS *(for now)*

Converting AST to Go Code



A1	Text	1
	Result	1.0

A2	Text	2
	Result	2.0

A3	Text	3
	Result	3.0

A4	Text	4
	Result	4.0

A5	Text	5
	Result	5.0

```
sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5, 7 - 6)`)

sheet.Calculate()
```

```
func (eng *pluginEngine) Parse(row, col int, ctx *model.Context) {
   cell := ctx.Cells.GetCell(row, col)
   cell.ParseAST()

   for _, compileInfo := range CompileFormula(row, col, cell.AST) {
       hashID := eng.addCompiledFormula(compileInfo)
       if hashID != `` {
           continue
       }
       cell.FID = compileInfo.ID
       eng.needsCompiled = true
   }
}
```

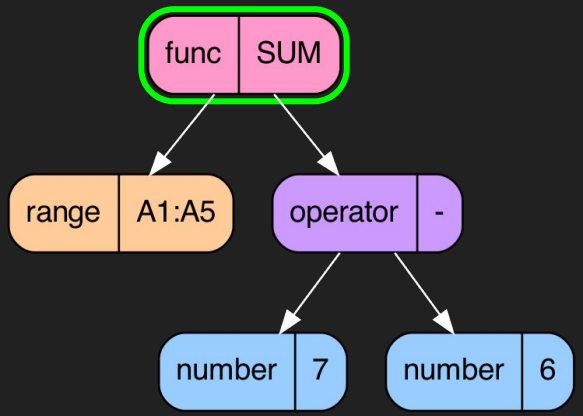
We use Postorder* Tree Traversal to transpile the AST

Postorder Algorithm

1. Traverse the left subtree
2. Traverse the right subtree(s)
3. Visit the root

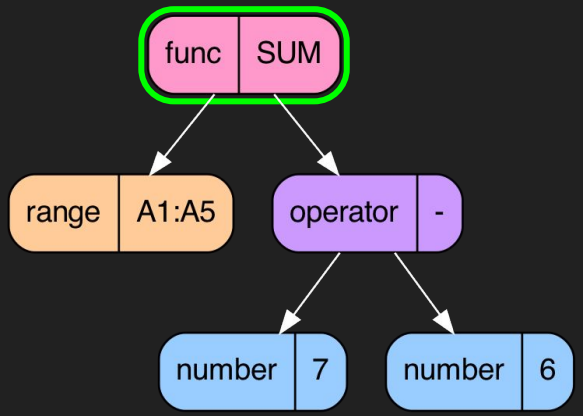
a.k.a. RPN (Reverse Polish Notation)

```
func compile(r, c int, node parser.Node, info *[]CompileInfo) string {
   switch n := node.(type) {
   case *parser.Bool:
       return fmt.Sprintf("B(%q)", n.Value)

   case *parser.Number:
       d, _ := strconv.Atoi(n.Value)
       return fmt.Sprintf("N(%d)", d)

   case *parser.String:
       return fmt.Sprintf("T(%q)", n.Value)

   case *parser.Operator:
   case *parser.Cell:
   case *parser.Range:
   case *parser.Function:
...

   }

   return ``
}
```



Transpiled Code

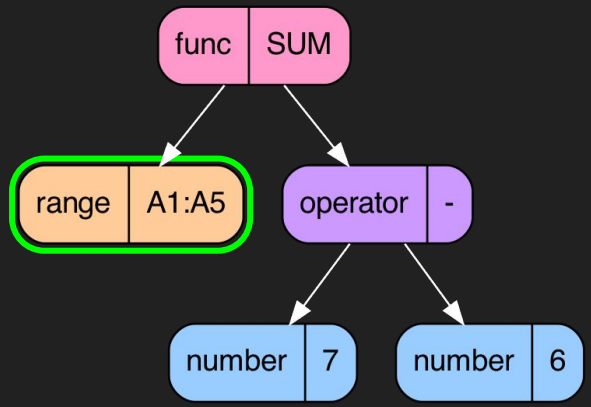
```
func compile(r, c int, node parser.Node, info *[]CompileInfo) string {
   switch n := node.(type) {
...
   case *parser.Function:
        funcName := strings.Title(strings.ToLower(n.Name))

        vals := make([]string, 0, len(n.Arguments))
        for _, arg := range n.Arguments {
            vals = append(vals, compile(r, c, arg, info))
        }

        args := strings.Join(vals, `,`)
        return fmt.Sprintf("x.%s(%s)", funcName, args)
```



Transpiled Code

```
func compile(r, c int, node parser.Node, info *[]CompileInfo) string {
   switch n := node.(type) {
...
   case *parser.Function:
       funcName := strings.Title(strings.ToLower(n.Name))

       vals := make([]string, 0, len(n.Arguments))
       for _, arg := range n.Arguments {
           vals = append(vals, compile(r, c, arg, info))
       }

       args := strings.Join(vals, `,`)
       return fmt.Sprintf("x.%s(%s)", funcName, args)
```

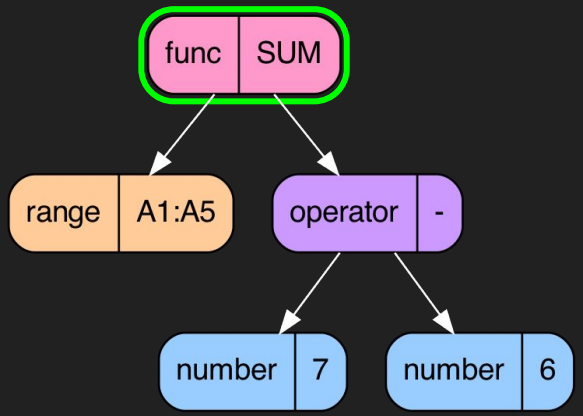
```
sheet.SetCell(`B1`, `=SUM( A1:A5, 7 - 6)`)
```



Transpiled Code

```
R(r,c-1,r+4,c-1)
```

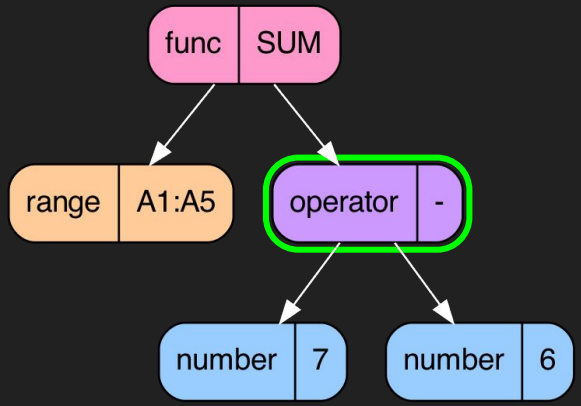
```
func compile(r, c int, node parser.Node, info *[]CompileInfo) string {
   switch n := node.(type) {
...
   case *parser.Range:
       parts := strings.Split(n.Value, `:`)

       row1, absRow1, col1, absCol1 := ToRowColWithAnchors(parts[0])
       rowRef1 := offset(`r`, r, row1, absRow1)
       colRef1 := offset(`c`, c, col1, absCol1)

       row2, absRow2, col2, absCol2 := ToRowColWithAnchors(parts[1])
       rowRef2 := offset(`r`, r, row2, absRow2)
       colRef2 := offset(`c`, c, col2, absCol2)

       return fmt.Sprintf(
           "R(%s,%s,%s,%s)",
           rowRef1, colRef1,
           rowRef2, colRef2,
       )
```



Transpiled Code

```
R(r,c-1,r+4,c-1)
```

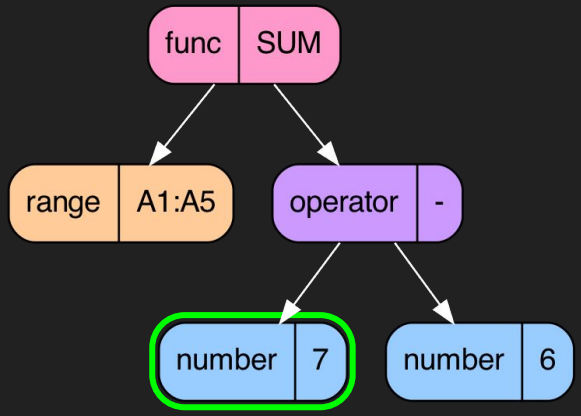
```
func compile(r, c int, node parser.Node, info *[]CompileInfo) string {
   switch n := node.(type) {
...
   case *parser.Function:
       funcName := strings.Title(strings.ToLower(n.Name))

       vals := make([]string, 0, len(n.Arguments))
       for _, arg := range n.Arguments {
           vals = append(vals, compile(r, c, arg, info))
       }

       args := strings.Join(vals, `,`)
       return fmt.Sprintf("x.%s(%s)", funcName, args)
```



Transpiled Code

```
R(r,c-1,r+4,c-1)
```

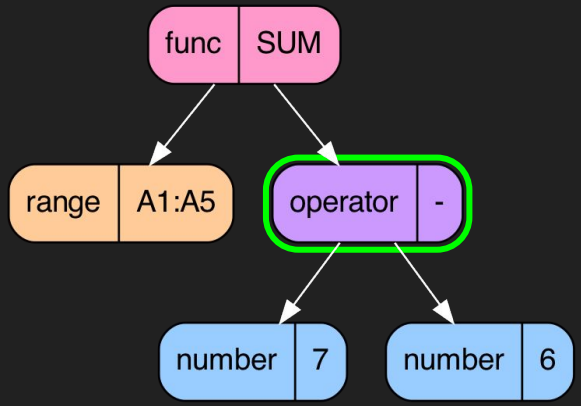
```
func compile(r, c int, node parser.Node, info *[]CompileInfo) string {
   switch n := node.(type) {
...
   case *parser.Operator:
       op := opNames[n.Operation]
       vals := make([]string, 0, 2)

       if n.LHS != nil {
           vals = append(vals, compile(r, c, n.LHS, info))
       }

       if n.RHS != nil {
           vals = append(vals, compile(r, c, n.RHS, info))
       }

       args := strings.Join(vals, `,`)
       return fmt.Sprintf("%s(%s)", op, args)
```




```
func compile(r, c int, node parser.Node, info *[]CompileInfo) string {
   switch n := node.(type) {
...
   case *parser.Number:
       d, _ := strconv.Atoi(n.Value)
       return fmt.Sprintf("N(%d)", d)
```

Transpiled Code

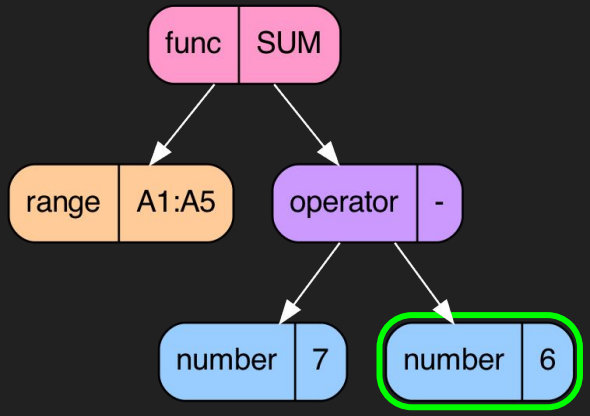
```
R(r,c-1,r+4,c-1)

N(7)
```



Transpiled Code

```
R(r,c-1,r+4,c-1)

N(7)
```

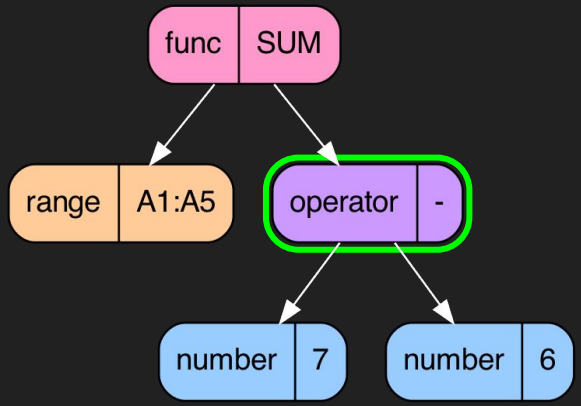
```
func compile(r, c int, node parser.Node, info *[]CompileInfo) string {
   switch n := node.(type) {
...
   case *parser.Operator:
       op := opNames[n.Operation]
       vals := make([]string, 0, 2)

       if n.LHS != nil {
           vals = append(vals, compile(r, c, n.LHS, info))
       }

       if n.RHS != nil {
           vals = append(vals, compile(r, c, n.RHS, info))
       }

       args := strings.Join(vals, `,`)
       return fmt.Sprintf("%s(%s)", op, args)
```




```
func compile(r, c int, node parser.Node, info *[]CompileInfo) string {
   switch n := node.(type) {
...
   case *parser.Number:
       d, _ := strconv.Atoi(n.Value)
       return fmt.Sprintf("N(%d)", d)
```

Transpiled Code

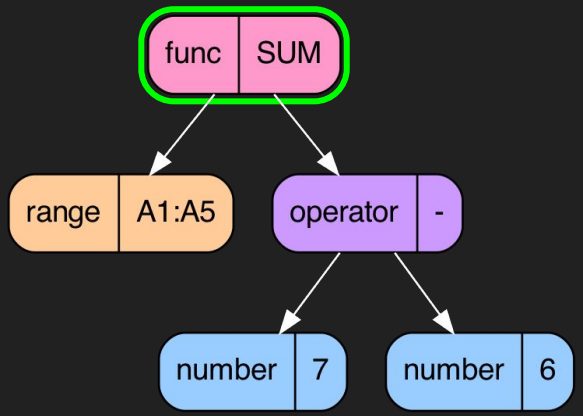
```
R(r,c-1,r+4,c-1)

N(7)
N(6)
```



Transpiled Code

```
R(r,c-1,r+4,c-1)

x.Sub(N(7),N(6))
```

```
func compile(r, c int, node parser.Node, info *[]CompileInfo) string {
   switch n := node.(type) {
...
   case *parser.Operator:
       op := opNames[n.Operation]
       vals := make([]string, 0, 2)

       if n.LHS != nil {
           vals = append(vals, compile(r, c, n.LHS, info))
       }

       if n.RHS != nil {
           vals = append(vals, compile(r, c, n.RHS, info))
       }

       args := strings.Join(vals, `,`)
       return fmt.Sprintf("%s(%s)", op, args)
```



Transpiled Code

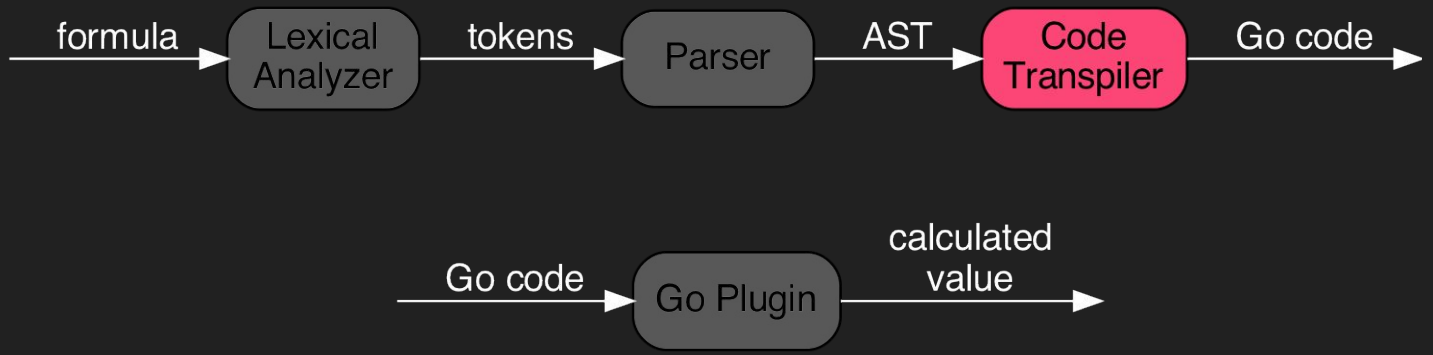
```
x.Sum(
    R(r,c-1,r+4,c-1),
    x.Sub(N(7),N(6)),
)
```

```
func compile(r, c int, node parser.Node, info *[]CompileInfo) string {
   switch n := node.(type) {
...
   case *parser.Function:
       funcName := strings.Title(strings.ToLower(n.Name))

       vals := make([]string, 0, len(n.Arguments))
       for _, arg := range n.Arguments {
           vals = append(vals, compile(r, c, arg, info))
       }

       args := strings.Join(vals, ",\n")
       return fmt.Sprintf("x.%s(\n%s,\n)", funcName, args)
```

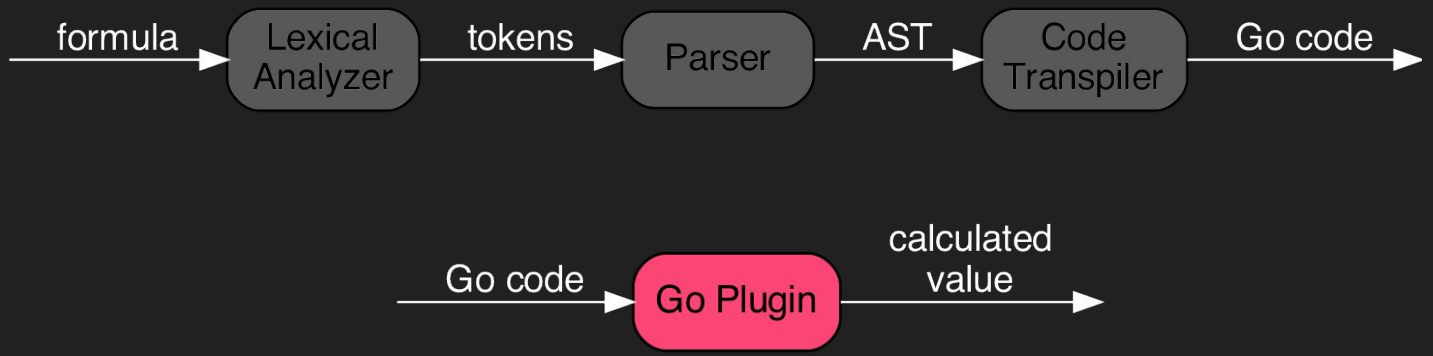
Final Output: compiled/gen.go

```
package main

import (
    . "github.com/therealtravissmith/gophercon-2020-talk/demo/compiled/helpers"
    m "github.com/therealtravissmith/gophercon-2020-talk/demo/model"
)

var _ = B // avoid "imported and not used" error

func Run(id string, r, c int, x *m.Context) (v *m.Result) {
    switch id {
    case `B1`:
        v = x.Sum(
            R(r, c-1, r+4, c-1),
            x.Sub(N(7), N(6)),
        )
    }
    return
}
```

How do we use our generated Go code?



Compiling and Running a Plugin



A1	Text	1
	Result	1.0

B1	Text	=SUM(A1:A5, 7 - 6)
	Result	empty
	AST	*parser.Node

A2	Text	2
	Result	2.0

A3	Text	3
	Result	3.0

A4	Text	4
	Result	4.0

A5	Text	5
	Result	5.0

```
sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5, 7 - 6)`)

sheet.Calculate()
```

```go
func (eng *pluginEngine) Calculate(row, col int, ctx *model.Context) *model.Result {
    cell := ctx.Cells.GetCell(row, col)
    if !cell.IsFormula {
        return cell.Result
    }
    if cell.Result.Type != model.ResultType_Empty {
        return cell.Result
    }

    // rebuild and load the plugin if any formulas have changed
    if eng.needsCompiled {
        WritePlugin("./compiled/gen.go", eng.compiledFuncs)
        BuildPlugin("./compiled/gen.go", "./compiled/plugin.so")
        eng.runCompiledFormula = LoadCompiled("./compiled/plugin.so")
        eng.needsCompiled = false
    }

    result := eng.runCompiledFormula(cell.FID, row, col, ctx)
    cell.SetResult(result)
    return result
}
```

```
func WritePlugin(dest string, funcs []string) {
   var code string

   if len(funcs) == 0 {
       code = strings.ReplaceAll(PluginTemplate, RunPlaceholder, "\treturn")
   } else {
       code = strings.ReplaceAll(SwitchTemplate, FunctionsPlaceholder, strings.Join(funcs, "\n"))
       code = strings.ReplaceAll(PluginTemplate, RunPlaceholder, code)
   }

   fd, _ := os.Create(dest)
   w := bufio.NewWriter(fd)
   w.WriteString(code)
   w.Flush()
   fd.Close()
}
```

```go
func BuildPlugin(src, dest string) string {
   args := []string{`build`, `-buildmode=plugin`, `-o`, dest, src}
   out, _ := exec.Command(`go`, args...).CombinedOutput()
   return string(out)
}
```

```go
func (eng *pluginEngine) Calculate(row, col int, ctx *model.Context) *model.Result {
    cell := ctx.Cells.GetCell(row, col)
    if !cell.IsFormula {
        return cell.Result
    }
    if cell.Result.Type != model.ResultType_Empty {
        return cell.Result
    }

    // rebuild and load the plugin if any formulas have changed
    if eng.needsCompiled {
        WritePlugin("./compiled/gen.go", eng.compiledFuncs)
        BuildPlugin("./compiled/gen.go", "./compiled/plugin.so")
        eng.runCompiledFormula = LoadCompiled("./compiled/plugin.so")
        eng.needsCompiled = false
    }

    result := eng.runCompiledFormula(cell.FID, row, col, ctx)
    cell.SetResult(result)
    return result
}
```

```go
func LoadCompiled(path string) model.RunnerFunc {
   plug, _ := plugin.Open(path)

   runSymbol, _ := plug.Lookup(`Run`)

   runner, ok := runSymbol.(func(s string, r, c int, ctx *model.Context) *model.Result)
   if !ok {
       panic(`cannot find the Run function`)
   }

   return runner
}
```

```go
func (eng *pluginEngine) Calculate(row, col int, ctx *model.Context) *model.Result {
    cell := ctx.Cells.GetCell(row, col)
    if !cell.IsFormula {
        return cell.Result
    }
    if cell.Result.Type != model.ResultType_Empty {
        return cell.Result
    }

    // rebuild and load the plugin if any formulas have changed
    if eng.needsCompiled {
        WritePlugin("./compiled/gen.go", eng.compiledFuncs)
        BuildPlugin("./compiled/gen.go", "./compiled/plugin.so")
        eng.runCompiledFormula = LoadCompiled("./compiled/plugin.so")
        eng.needsCompiled = false
    }

    result := eng.runCompiledFormula(cell.FID, row, col, ctx)
    cell.SetResult(result)
    return result
}
```

```
func Run(id string, r, c int, x *m.Context) (v *m.Result) {
   switch id {
   case `B1`:
       v = x.Sum(
           R(r, c-1, r+4, c-1),
           x.Sub(N(7), N(6)),
       )
   }
   return
}
```

A1		
	Text	1
	Result	1.0

B1		
	Text	=SUM(A1:A5, 7 - 6)
	Result	empty
	AST	*parser.Node

A2		
	Text	2
	Result	2.0

A3		
	Text	3
	Result	3.0

A4		
	Text	4
	Result	4.0

A5		
	Text	5
	Result	5.0

```
sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5, 7 - 6)`)

sheet.Calculate()
```

A1		
	Text	1
	Result	1.0

B1		
	Text	=SUM(A1:A5, 7 - 6)
	Result	16.0
	AST	*parser.Node

A2		
	Text	2
	Result	2.0

A3		
	Text	3
	Result	3.0

A4		
	Text	4
	Result	4.0

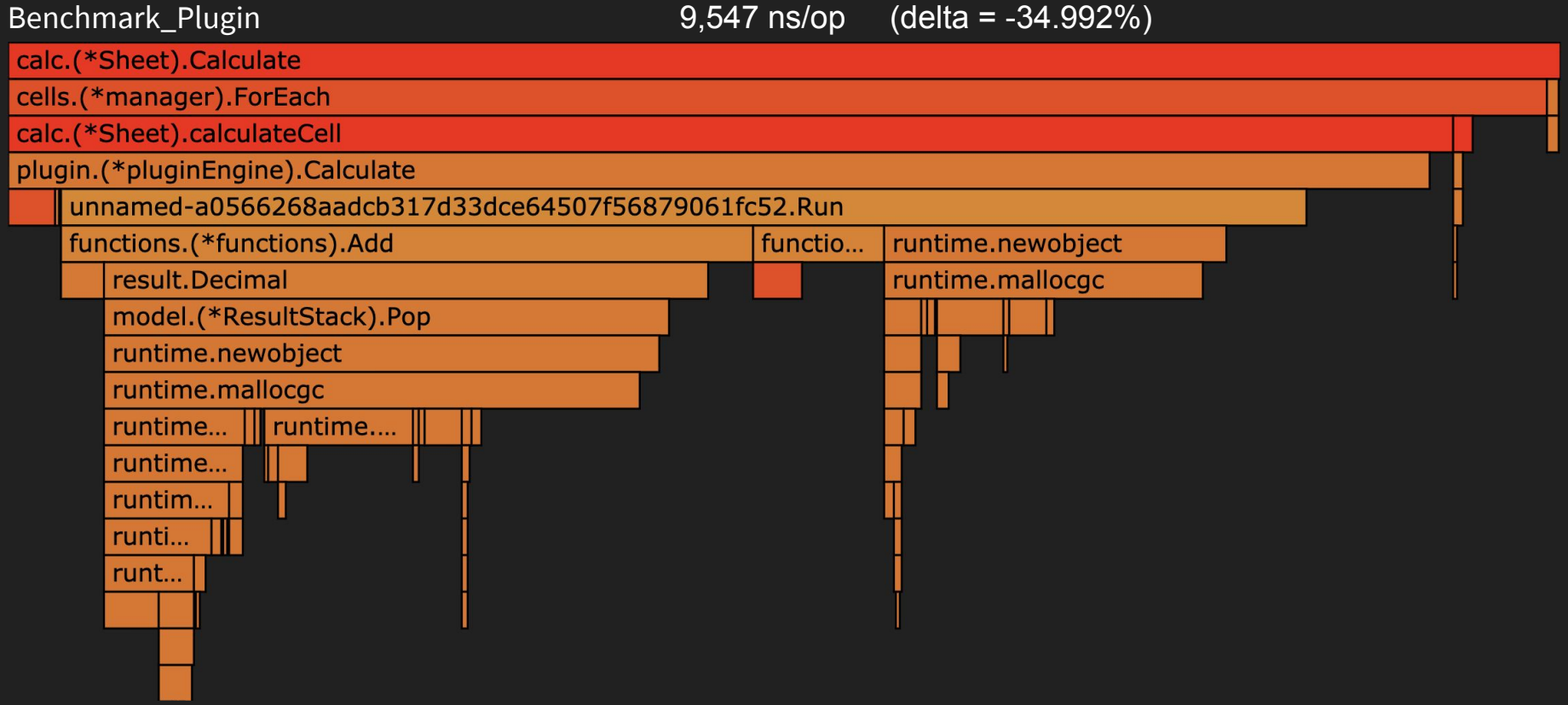
A5		
	Text	5
	Result	5.0

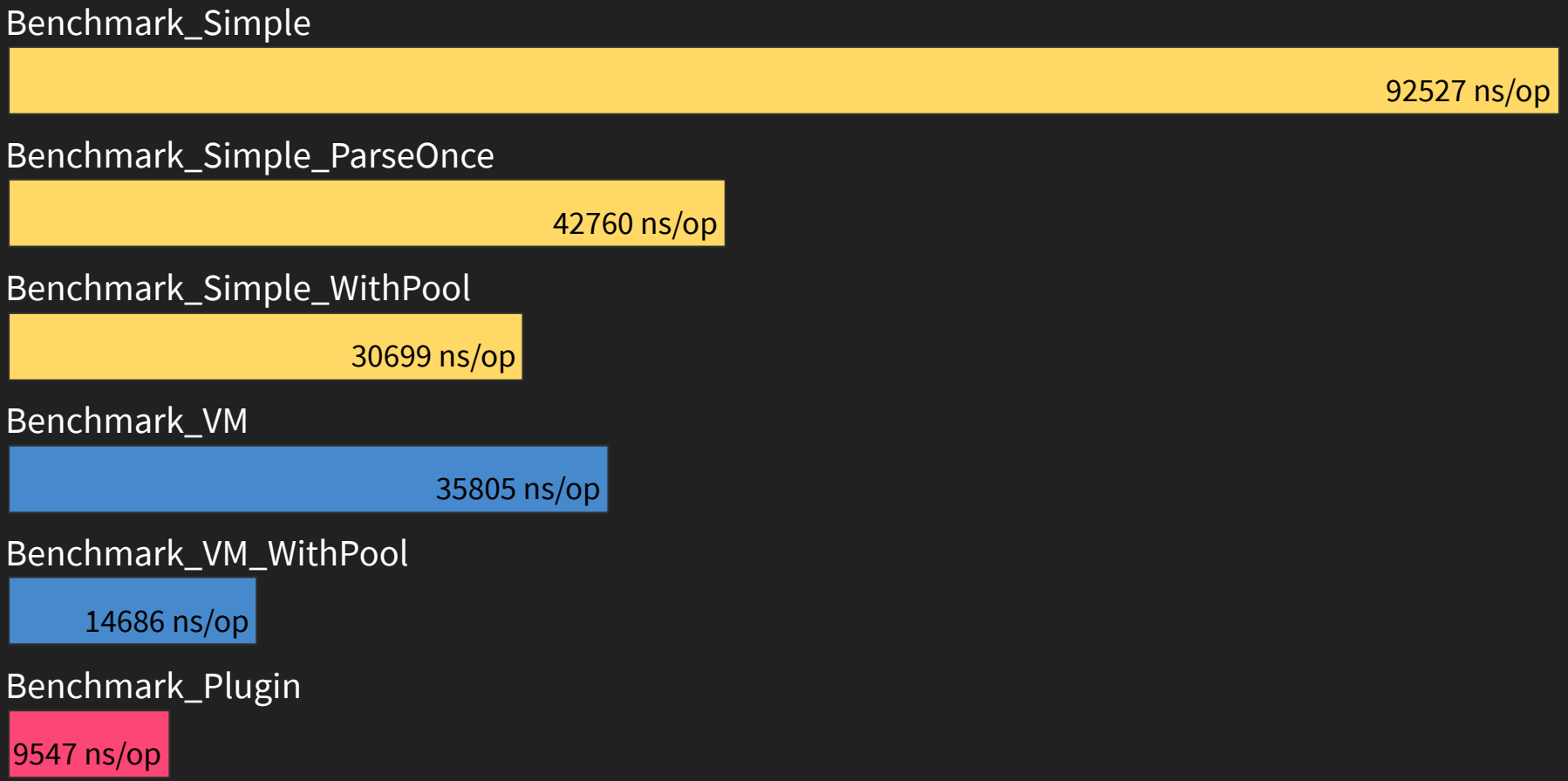
```
sheet.SetCell(`A1`, `1`)
sheet.SetCell(`A2`, `2`)
sheet.SetCell(`A3`, `3`)
sheet.SetCell(`A4`, `4`)
sheet.SetCell(`A5`, `5`)

sheet.SetCell(`B1`, `=SUM( A1:A5,

sheet.Calculate()
```

What about performance?

```
go build
./demo -engine=plugin -cpuprofile=plugin.cpu
go tool pprof plugin.cpu
```

Calculating with Go Plugins




Benchmark_Simple
92527 ns/op
Benchmark_Simple_ParseOnce
42760 ns/op
Benchmark_Simple_WithPool
30699 ns/op
Benchmark_VM
35805 ns/op
Benchmark_VM_WithPool
14686 ns/op
Benchmark_Plugin
9547 ns/op

-89.68% of original calculation time!

Go Plugins for the Win!

But can we do even better?

But can we do even better?

YES! But I'll save that for another time...

One warning though...

One warning though...

- `WritePlugin()` completed in 205 µs
- `BuildPlugin()` completed in 729 ms
- `LoadCompiled()` completed in 164 ms

One warning though...

- `WritePlugin()` completed in 205 µs
- `BuildPlugin()` completed in 729 ms
- `LoadCompiled()` completed in 164 ms

Creating plugins is extremely slow, using plugins is extremely fast!

Lessons Learned

How to Improve Performance

- Look for ways to do work as few times as possible

How to Improve Performance

- Look for ways to do work as few times as possible
- Use object pooling to reduce runtime memory allocations

How to Improve Performance

- Look for ways to do work as few times as possible
- Use object pooling to reduce runtime memory allocations
- Create a VM if you think the added performance is worth it

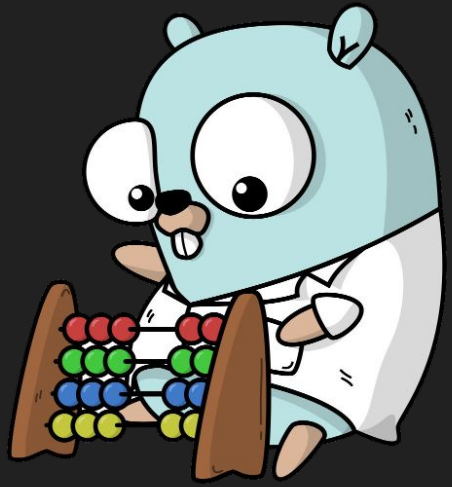
How to Improve Performance

- Look for ways to do work as few times as possible
- Use object pooling to reduce runtime memory allocations
- Create a VM if you think the added performance is worth it
- Use Go plugins if you need the absolute speed and can amortize the compile time

Creating 250 slides takes a *really* long time

Optimizing Performance

using a VM and Go Plugins



Travis Smith
Senior Software Architect at Workiva
GopherCon 2020