Министерство науки и высшего образования Российской Федерации

ФГБОУ ВО «АЛТАЙСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт цифровых технологий, электроники и физики (ИЦТЭФ)

Кафедра вычислительной техники и электроники (ВТиЭ)

Отчет по проектно-технологической практике

(Научно-исследовательская работа)

РАЗРАБОТКА ЯЗЫКА ПРОГРАММИРОВАНИЯ:

ЛЕКСИЧЕСКИЙ И СИНТАКСИЧЕСКИЙ АНАЛИЗАТОРЫ

Выполнил студент 506 группы:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Д.В. Осипенко

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_ 2023 г.

Проверил: ст. преп. кафедры ВТиЭ.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ П.Н. Уланов

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_ 2023 г.

РЕФЕРАТ

В данной работе рассматривается первая ступень компилятора, в частности первые два этапа: лексический и синтаксический анализы. Впоследствии происходит разработка лексического и синтаксического анализатора для простого языка программирования, определенного с помощью расширенной формы бэкуса-наура.

Ключевые слова: ПАРСЕР, СКАНЕР, LL-ПАРСЕР, LR-ПАРСЕР, СИНТАКСИЧЕСКОЕ ДЕРЕВО, EBNF.

**СОДЕРЖАНИЕ**

[**Введение 4**](#_539pho8uv2vx)

[**1. Краткий обзор компиляторов. 5**](#_7r3zhykbgrty)

[1.1. Устройство компилятора. 5](#_xhwr8mokilio)

[1.2. Основные разновидности компиляторов. 5](#_oscykc6p1g4j)

[**2. Обзор лексического и синтаксического анализаторов. 7**](#_yu96hycdsvq2)

[2.1. Устройство лексического анализатора 7](#_n3jtizfsy431)

[2.2. Устройство парсера 9](#_a7supgd845gz)

[**3. Разработка компилятора. 11**](#_80euvyqypld5)

[3.1. Постановка задачи. 11](#_4bxpysn2kmke)

[3.2. Лексический анализатор. 12](#_9edjss6fl4d6)

[3.3. Синтаксический анализатор 17](#_hsbhlje38kpa)

[**Заключение 26**](#_diutxh2gbnyc)

[**Список использованной литературы 27**](#_wgjbpd8v0rn5)

[**Приложение А. Текст программы “Лексер”. 28**](#_c1ic41hvfz6t)

[**Приложение Б. Текст программы “Парсер”. 36**](#_ush0xf15095q)

# Введение

На сегодняшний день любая программа написана с помощью какого-либо языка программирования (далее ЯП), от самого низкоуровневого, например ассемблер, до высокоуровневых представителей, таких как: Python, Ruby, Go, Rust, C/C++ и т.д.. Сам по себе ЯП представляет собой:

* Спецификацию;
* Программу компилятор или интерпретатор.

Компилятор являются главным элементом любого ЯП, определяющее основные показатели языка, как: быстродействие, поддерживаемые платформы и функциональные возможности. В следствии чего это направление можно назвать одним из основополагающих и до сих пор актуальных.

В данной научно-исследовательской работе (далее НИР) объектами изучения являются первые два этапа работы компилятора: лексический и синтаксический анализы. Будет рассмотрен разбор программы, основанной на грамматике (правилах) формального языка, заданного с помощью расширенной формы Бэкуса — Наура (далее EBNF).

Задачи НИР:

1. Поверхностный обзор устройства компилятора;
2. Знакомство с лексическим анализом;
3. Знакомство с синтаксическим анализом;
4. Разработка лексического анализатора и парсера для простого языка программирования.

# Краткий обзор компиляторов.

## Устройство компилятора.

Компилятора – это большое, сложное программное обеспечение (ПО). Компьютерное сообщество разрабатывает компиляторы начиная с 1955 г., и с каждым последующим годом мы все лучше узнавали как следует их проектировать [1].

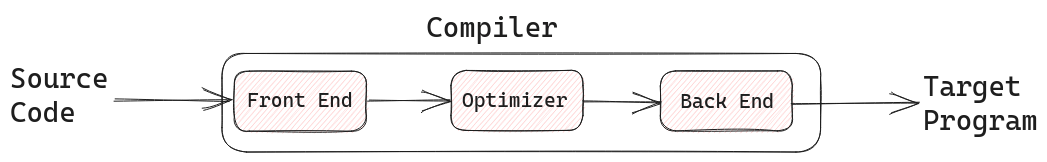


Рис 1.1 Оптимизирующий компилятор.

На сегодняшний день, структуру компилятора можно представить в виде трехфазной системы, также известную как оптимизирующий компилятор (рис 1.1). На практике, каждая из фаз поделена на некоторое количество промежуточных этапов. Например, **Front end** состоит обычно из 2-3 этапов, которые собирают информацию и детали из распознавания поступающего на вход исходного кода и преобразуя его в промежуточное представление (**IR**) программы. **Middle end** (или **Optimizer**) содержит этапы, которые осуществляет различные преобразования и оптимизацию. Количество этих этапов разнится от компилятора к компилятору. Ну и наконец, **Back end** состоит из серии этапов, каждый из которых приближает IR к набору инструкций целевой архитектуры [1].

## Основные разновидности компиляторов.

Компилятор преобразует высокоуровневый исходные код в равноценную низкоуровневую целевую программу (обычно в машинном языке), затем завершает свою работу. В любое время после, пользователь может сказать операционной системе выполнить данную скомпилированную программу [2]. Компилятор осуществляет только преобразование одного набора инструкций в другой, все последующее время программа существует и работает независимо от него. Данный подход называется транспилирующим компилятором (transpiler), или просто компилятор.

Альтернатива представленному выше компилятору для высокоуровневых языков известна как интерпретатор (interpreter). В отличии от первого, интерпретатор всегда осуществляет свою работу во время выполнения программы. Он последовательно считывает инструкции программы, выполняет их и завершает работу [2]. Графики примеров работы этих двух компиляторов можно увидеть на рис. 1.2.

# 

Рис 1.2 Принцип работы: a)Компилятора; b) Интерпретатора.

# Обзор лексического и синтаксического анализаторов.

## Устройство лексического анализатора

Front end компилятора начинается с потока символов, которые в совокупности представляют текст программы, и ожидается, что их них будет создано промежуточное представление, позволяющее вместить в себя всю важную информацию и преобразовать его в эффективный и верный целевой код [3].

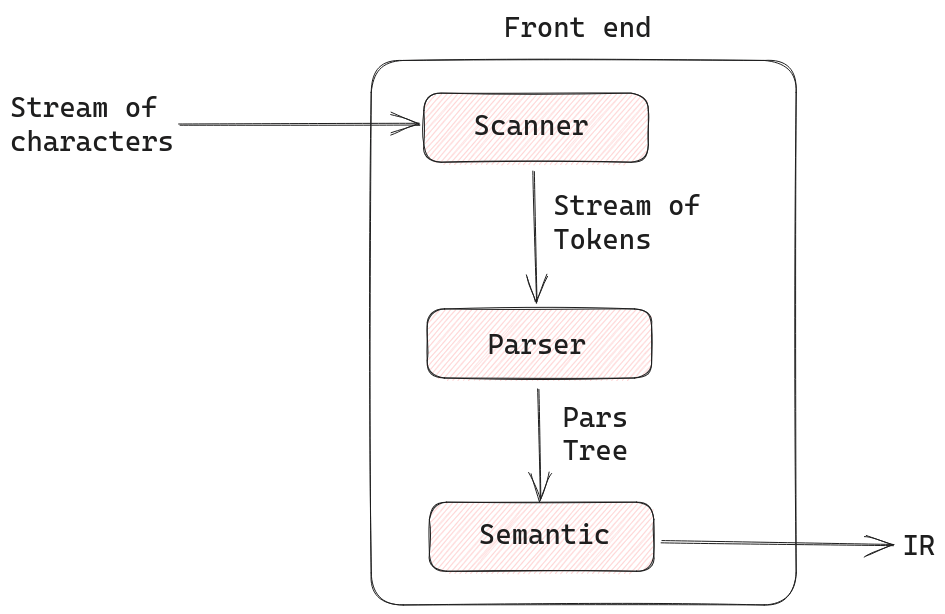


Рис. 2.1 Трехступенчатая структура первого этапа (Front end) компилятора.

Сканнер (или лексический анализатор) – первая из 3 ступеней, изображенных на рисунку 2.1, которые компилятор использует для понимания входной программы. Сканер считывает поток символов и группируя их по определенным правилам, создает поток слов (токенов), где каждое слово относится к определенной категории (символы, строки, числа, и т.п.) [1].

Каждый тип токена определен регулярным выражением. Некоторые токены имеют заранее известный размер и соответствующие простые регулярные выражения: ‘:’, ‘=’, ‘,’ ‘/’. Другие, в свою очередь, могут иметь неопределенный размер, такие как: наименования, строки, числа, коментарии. Ключевые и зарезервированные слова, в свою очередь, могут относится к первой разновидности токенов, но намного удобней, с практической точки зрения, относить их ко второй. Совокупность типа токена и определяющего его регулярного выражения называют - описанием токена, которое можно представить в следующем виде [3]:

assign\_token => ‘=’ ;

colon\_token => ‘:’ ;

. . .

while\_token => ‘while’;

for\_token => ‘for’;

. . .

string\_token => ‘“’[a-zA-Z0-9]\*’”’;

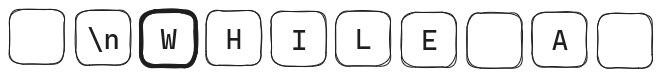
identifier\_token => [a-zA-Z]+[a-zA-Z\_]\*;

Где слева находится тип токена, а справа его регулярное выражение.

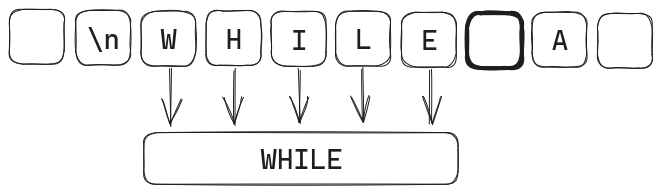
Но как сканер будет проходить, сканировать строку и впоследствии извлекать токен? представим мы имеем строку



Сканер начинает считывать ее с 0 позиции пропуская все незначащие символы до тех пор, пока не найдет важные для себя символ. В нашем случае он остановится на букве ‘W’:



Так как это буква, сканер предполагает что это слово и начинает считывать его в промежуточный буфер вплость до первого пустого символа:



В буфер записалось значение “while”, которое проверяется в списке зарезервированных слов, если оно там содержится, то это ключевое слово, иначе это просто наименование. Данный процесс повторяется до тех пор, пока сканер не дойдет до конца файла или строки или не обнаружит ошибку, не прописанное правило [4].

Сканеры в реальных компиляторах обычно пишут в ручную, код для них получается простым и малым по размеры. Но во время разработки бывает удобно представить его в неком стороннем представлении, таком как конечные автоматы, показанном на рисунке 2.2 [2].

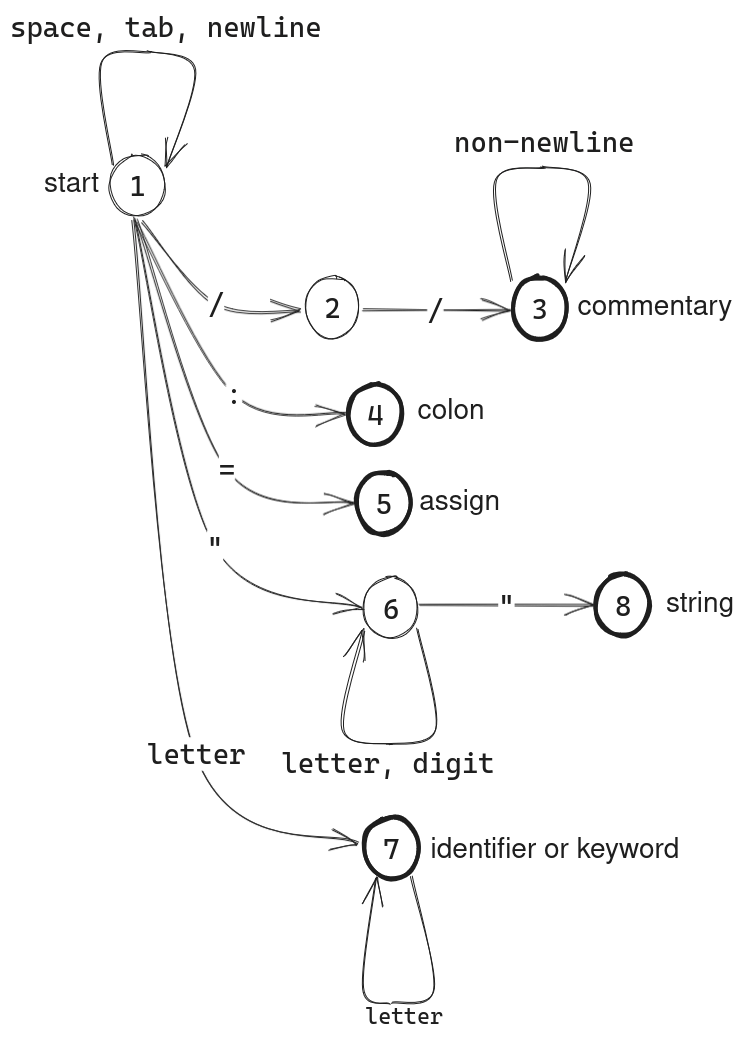


Рис. 2.2 Изображение сканера в форме конечного автомата.

## Устройство парсера

Парсер – это вторая ступень на этапе распознавания текста программы. Он работает с потоком токенов, полученных с предыдущей ступени – сканера. Основные задачи парсера: сгруппировать наборы токенов в синтаксические структуры, которые, в свою очередь, образуют синтаксическое дерево; определить синтаксическую верность поступающей на вход программы. В общем случае парсеры усиленно изучались как часть теории формальных языков. Эти наработки используются как теоретический базис для практических способов реализации парсеров, применяемых в настоящих промышленных компиляторах [1][2][3].

Существует два наиболее известных и хорошо изученных метода построение парсеров:

1. слева-направо сверху-вниз, известный как LL (**L**eft-to-right **L**eft-most derivation);
2. слева-направо снизу-вверх, известные как LR (**L**eft-to-right **R**ight-most derivation).

Слева-направо значит что текст программы, или набор токенов, обрабатываются начиная с самого левого токена и заканчивая самым правым, по одному токену за раз. Сверху-вниз означает, что парсер начинает построение дерева с его корня стремясь к его листьям, в то время как снизу-вверх начинает с листьев, стремясь к его корню [3].

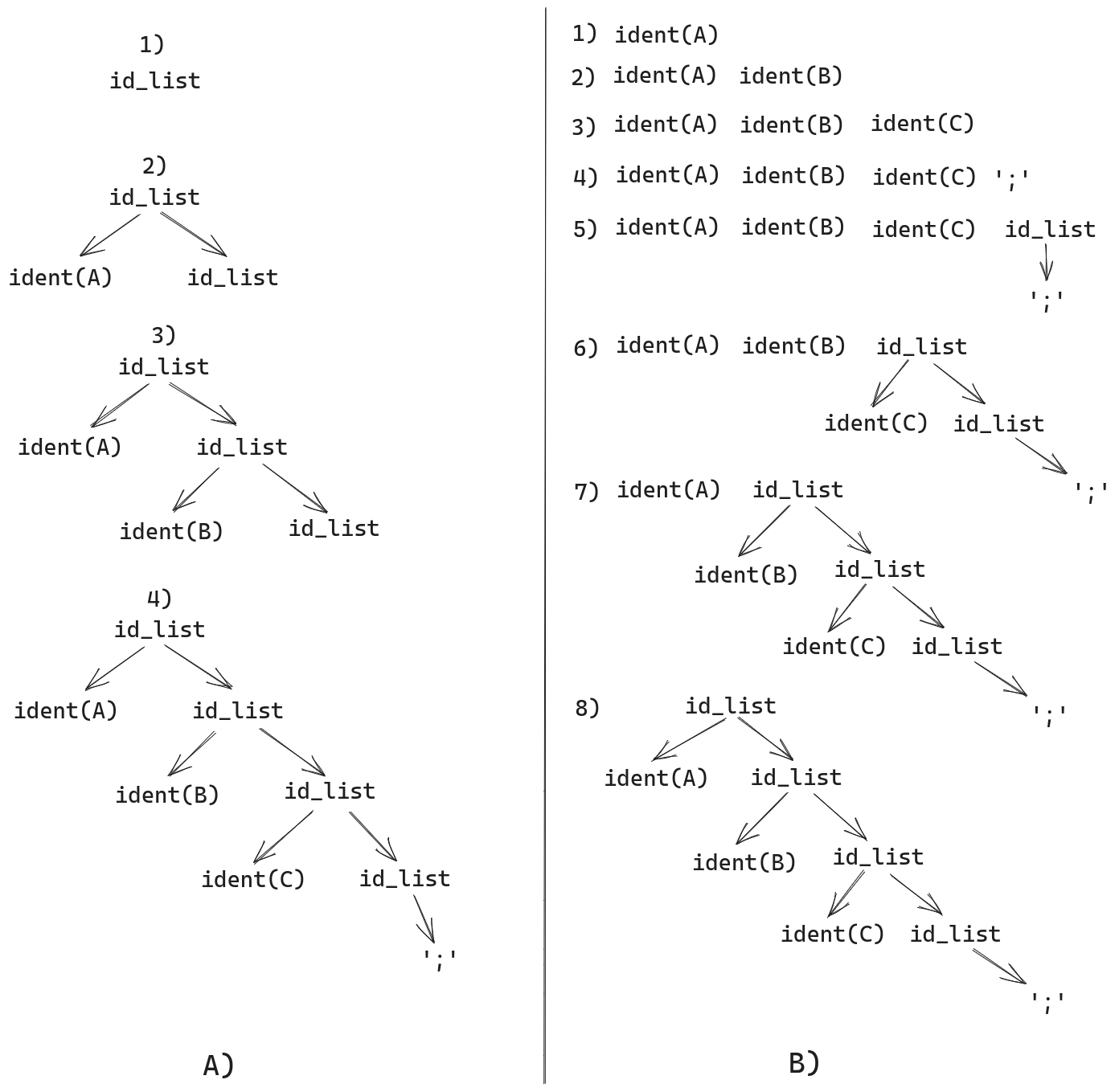


Рис. 2.3 Пример парсинга выражения **‘A B C ;’** с помощью парсера:

A) LL; B) LR.

Для наглядного представления разницы работы двух видов парсеров, разработаем простую грамматику для списка наименований:

id\_list => ident id\_list ;

| ‘;’ ;

Пример разложения выражения ‘ **A B C ;** ’ с использованием двух методов представлен на рисунке 2.3.

# Разработка компилятора.

## Постановка задачи.

Перед началом разработки любого ПО необходимо установить некоторые требования и задачи, стоящие перед ним. К компилятору это относится в особенной степени, ведь от работоспособности данного ПО зависит, как не странно, работоспособность всего остального ПО, производимое им. Так как это всего лишь учебный проект, то и требования к нему будут существенно меньше чем к промышленным собратьям и они таковы:

**Задача** – Разработать лексический и синтаксический анализаторы, на основе приведенных ниже регулярных выражений и грамматических структур. Проверить быстродействие и правильность результатов выполнения анализаторов с помощью нескольких примеров программ.

**Регулярные выражения:**

SYMBOL => [a-zA-Z]+[a-zA-Z\_]\* ;

NUMBER => [0-9]+[.][0-9]\* ;

STRING => ‘“‘ [a-zA-Z0-9]\* ’”’ ;

TRUE => ‘true’ ;

FALSE => ‘false’ ;

CONST => ‘@const’ ;

SET => ‘@set’ ;

WHILE => ‘@while’ ;

VAR => ‘@var’ ;

FN => ‘@fn’ ;

IF => ‘@if’ ;

ELSE => ‘else’ ;

**Грамматические структуры** в EBNF:

programNode => globals\* ;

atom => SYMBOL | STRING | NUMBER | TRUE | FALSE ;

globals => constNode | varNode | fnNode ;

locals => constNode | setNode | varNode | ifNode | whileNode | exprNode ;

constNode => CONST SYMBOL ‘:’ SYMBOL ‘=’ atom ‘;’ ;

setNode => SET SYMBOL ‘=’ exprArgNode ‘;’ ;

varNode => VAR SYMBOL ‘:’ SYMBOL ‘=’ exprArgNode ‘;’ ;

fnNode => FN SYMBOL ‘:’ SYMBOL ‘(‘ (SYMBOL ‘:’ SYMBOL)\* ‘)’ ‘{‘ locals\* ‘}’ ;

ifNode => IF exprArgNode ‘{‘ locals+ ‘}’ (ELSE ‘{‘ locals+ ‘}’)? ;

whileNode => WHILE exprArgNode ‘{‘ locals + ‘}’ ;

exprNode => ‘(‘ SYMBOL exprArgNode\* ‘)’ ;

exprArgNode => exprNode | atom ;

Для разработки компилятора будет использоваться язык программирования Go (Golang) [5] от компании Google выпущенный в 2009 году. Причиной для выбора можно назвать как личные предпочтения, возможные перспективы, и баланс между сложностью, удобством разработки и быстродействием относительно других представителей рынка, таких как Java, C/C++, JavaScript, Python.

## Лексический анализатор.

Лексический анализатор - это программа, которая на вход принимает текст программы, анализирует каждый ее символ, группирует их по определенным правилам, и на выходе возвращает последовательность токенов. Для распознавания и группировки существует несколько подходов: используя регулярные выражения или писать алгоритмы для каждого символа и отдельных вариаций. Для нашего компилятора был выбран второй вариант, так как он наиболее эффективен по быстродействию, потреблению памяти и позволяет создавать более гибкие сообщения об ошибках чем первый вариант, но его сложнее и дольше писать.

Для начала нам необходимо определить токен, для этого мы создаем одноименную структуру, которая содержит в себе тип токена, его значение, и положение в тексте анализируемой программы:

type Token struct {

Type TokenType

Value string

Line int

Column int

}

Где типом токена является перечисление, каждый элемент которого соответствует типу uint8:

type TokenType uint8

const (

TokenIllegal TokenType = iota

TokenEOF

// literals

TokenNumber

TokenString

TokenSymbol

. . .

)

После этого мы можем приступить к созданию самого анализатора, который будет представлять из себя структуры, содержащую: текст программы, список токенов, список ошибок, индекс для текста программы, и положение относительно текста программы:

type Lexer struct {

input string

tokens []tokens.Token

errors []error

pos int

line int

column int

}

Лексер имеет несколько вспомогательных методов для упрощения разработки:

* next() – для получения следующего символа и увеличения внутреннего индекса на 1;
* peek(int) – позволяющий получить символ на некоторой позиции относительно текущего индекса;
* isEOF() – проверяет, указывает ли индекс на конец анализируемого текста;
* isChar(byte) – проверяет, является ли аргумент буквой;
* isDigit(byte) – проверяет, является ли аргумент цифрой;
* isPrintable(byte) – проверяет, является ли аргумент отображаемым символом;
* Error() – возвращает список ошибок.

Основополагающим методом является Lex() в котором и происходит основной анализ всего текста программы. Состоит он из одного цикла, внутри которого располагается огромный список условий, проверяющих каждый символ на потенциальную принадлежность к определенной группе токенов. Алгоритм работы данного метода можно описать так:

НАЧАЛО.

1. Извлечь символ из текста программы.

2. Проверить, есть ли определенные правила для данного символа, иначе вывести ошибку о неопознаном символе и перейти в п. 4.

3. Создать токен по соответствующим правилам

4. Проверить, является ли следующий символ концом файла, если да, то завершить работу, иначе указывать на следующий символ и перейти в п. 1.

КОНЕЦ.

Для проверки работоспособности программы возьмем лексически верный текст и подадим его на вход анализатора:

@const pi:float = 3.14;

@fn main:void() {

@var x:int = 2;

@set x = (floatToInt (addf (intToFloat x) pi))

@if (eq x 10) {

(println (std/intToString x))

}

}

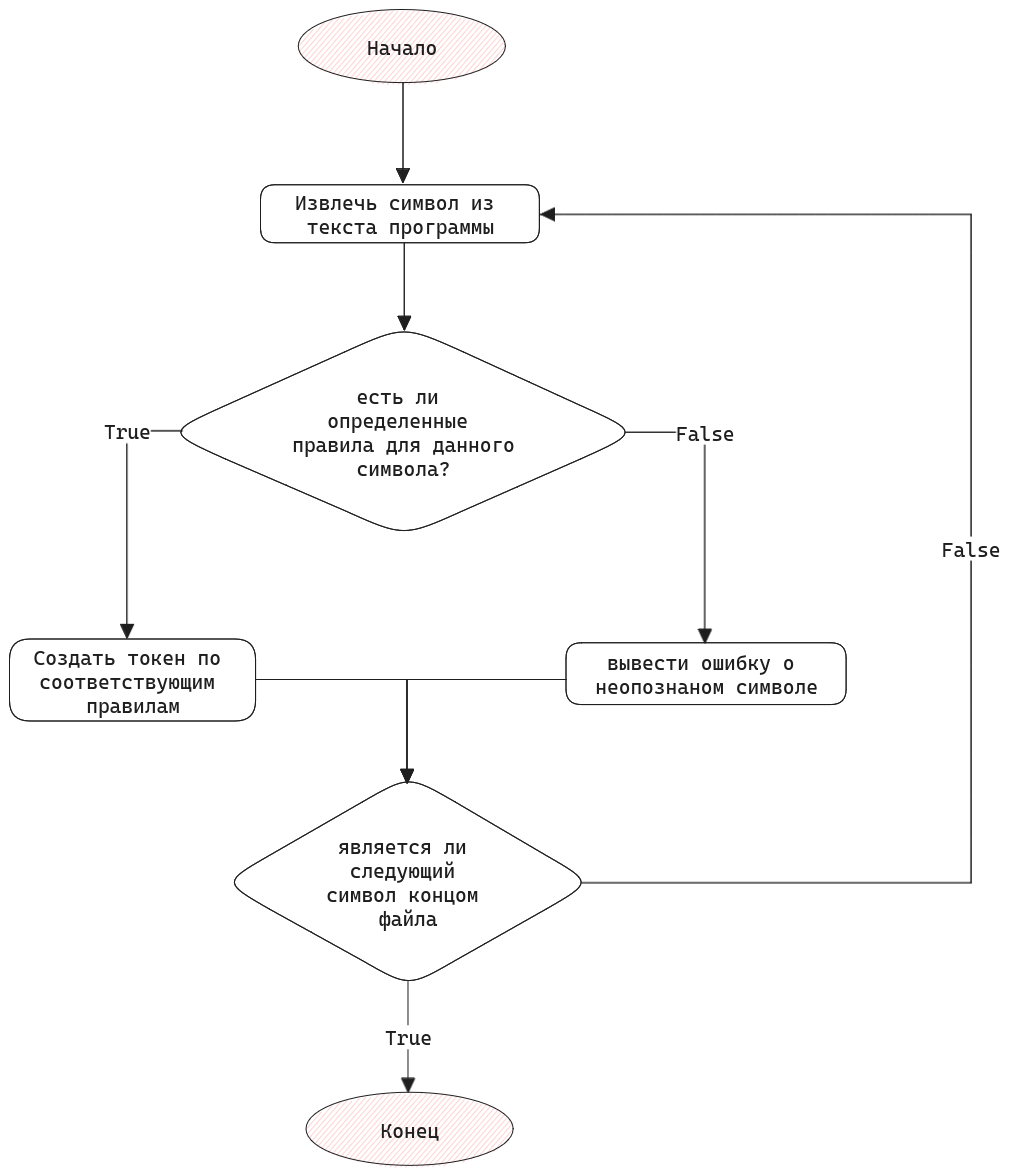


Рис. 3.1. Блок схема алгоритма лексического анализатора.

На выходе мы получаем последовательность токенов:

(Type @const, Value @const)

(Type SYMBOL, Value pi)

(Type :, Value :)

(Type SYMBOL, Value float)

(Type =, Value =)

(Type NUMBER, Value 3.14)

(Type ;, Value ;)

(Type @fn, Value @fn)

(Type SYMBOL, Value main)

(Type :, Value :)

(Type SYMBOL, Value void)

(Type (, Value ()

(Type ), Value ))

(Type {, Value {)

(Type @var, Value @var)

(Type SYMBOL, Value x)

(Type :, Value :)

(Type SYMBOL, Value int)

(Type =, Value =)

(Type NUMBER, Value 2)

(Type ;, Value ;)

(Type @set, Value @set)

(Type SYMBOL, Value x)

(Type =, Value =)

(Type (, Value ()

(Type SYMBOL, Value floatToInt)

(Type (, Value ()

(Type SYMBOL, Value add)

(Type (, Value ()

(Type SYMBOL, Value intToFloat)

(Type SYMBOL, Value x)

(Type ), Value ))

(Type SYMBOL, Value pi)

(Type ), Value ))

(Type ), Value ))

(Type @if, Value @if)

(Type (, Value ()

(Type SYMBOL, Value eq)

(Type SYMBOL, Value x)

(Type NUMBER, Value 10)

(Type ), Value ))

(Type {, Value {)

(Type (, Value ()

(Type SYMBOL, Value println)

(Type (, Value ()

(Type SYMBOL, Value std)

(Type /, Value /)

(Type SYMBOL, Value intToString)

(Type SYMBOL, Value x)

(Type ), Value ))

(Type ), Value ))

(Type }, Value })

(Type }, Value })

(Type EOF, Value )

Результаты замеров производительности при разном количестве входных данных ( в качестве данных использовались результаты генератора Lorem ipsum) представлены на рисунки 3.2. Видно, что зависимость среднего затраченного времени на операцию за 1000 итераций при различных объемах входного текста выражается практически линейно.

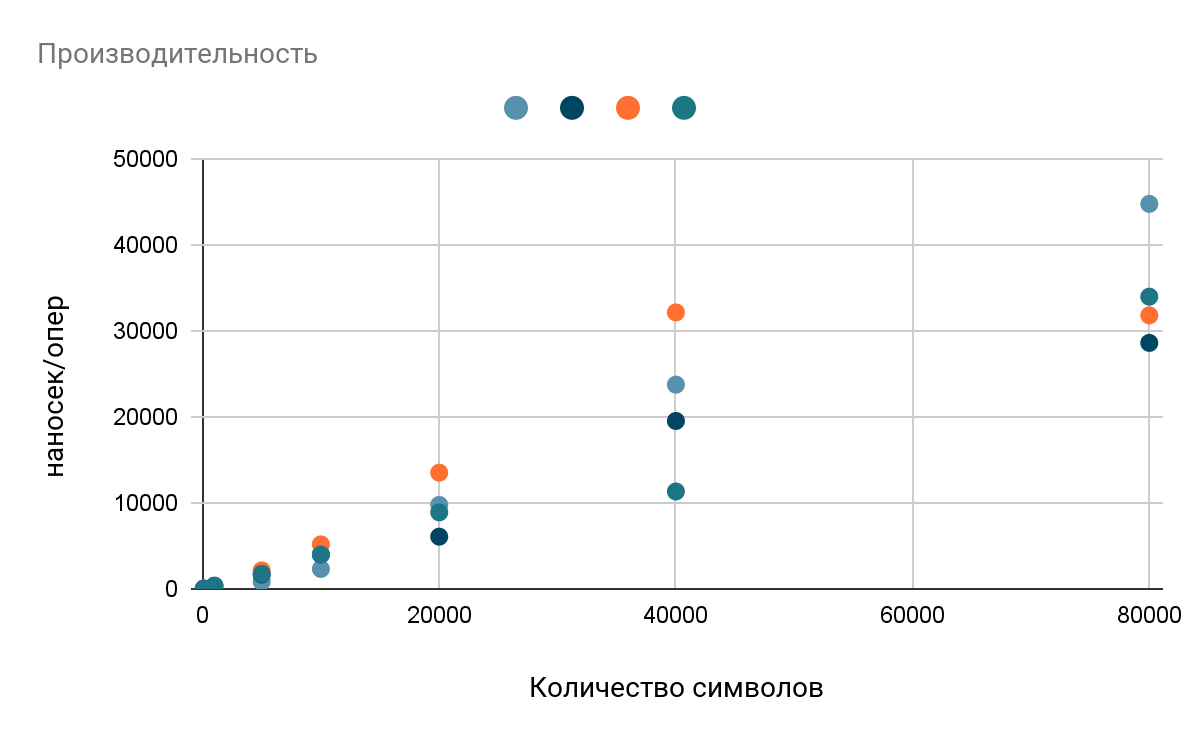


Рис. 3.2 Зависимость среднего затраченного времени на операцию за 1000 итераций при различном размере входных данных.

## Синтаксический анализатор

После производства последовательности токенов в дело вступает синтаксический анализ – процесс группировки токенов по группам на основе предопределенной грамматики. Для разработки парсера можно прибегнуть к нескольким способам:

* обратить свой взор в сторону различных генераторов, таких как bison [6], antlr [7];
* воспользоваться сторонними библиотеками, например participle [8];
* написать парсер вручную.

В первом случае вы задаете файл, в котором описываются грамматические конструкции, после он передается соответствующей программе, которая генерирует представления структур на целевом языке. Плюсами данного подхода можно отнести гибкость, быстроту разработки и возможность генерировать код на разных языках при одной и той же структуре грамматического файла. К основным минусам часто относят невозможность задать более понятные сообщения об ошибках и не самая высокая производительность генерируемого кода, относительно ручных вариантов [9].

Относительно сторонних библиотек, они могут решить проблему с производительностью, т.к. библиотека уже зарание заточена на генерацию кода на конкретном языке, но также страдают сообщения об ошибках. Participle прекрасная библиотека для создания собственных парсеров (и лексеров) за короткий промежуток времени, достигая близкой к самописным экземплярам производительности. Если суммировать, то при наличии целей создание своего парсера на языке программирования Go это хороший вариант.

Далее будет использоваться третий подход – написание парсера вручную. Причины на это довольно просты: чтобы чему-то научится и понять, это что-то нужно сделать; можно добится большей производительности; большая гибкость в задании грамматики и ее реализации, которые не способны быть исполнены в генераторах; возможность написать более внятные сообщения об ошибках. К главному минусу данного подхода относится времязатратность – чтобы разработать парсер вручную, придется прописывать каждое правило, что может сильно затянутся, но, как видно из плюсов, обмен абсолютно равноценен.

Для разработки самописного парсера прибегают к LL(\*), под названием Рекурсивный Спуск (Recursive Descent). Это самый популярный самописный парсер среди промышленных компиляторов, который используют такие языки как C/C++, Python, Go. Суть этого парсера проста, для каждого правила присутствует свой метод, которые рекурсивно используются парсером без прибегания к возврату в прошлое состояние. Также этот парсер называют предсказывающим, так как он заранее знает что должно следовать дальше основываясь на входящие данные.

Прежде чем приступить к разработке парсера, необходимо определить основную рабочую единицу – структуру синтаксического дерева. В ходе экспериментов и многократного переписывания кода, была выведен следующий подход к построению синтаксического дерева:

type Node interface {

json.Marshaler

node()

}

Node – это интерфейс, вмещающей в себе все составляющие нашего дерева. Для того, чтобы какая-либо структура могла быть частью дерева, она должна реализовывать в себе представленный выше интерфейс. В принципе этого достаточно чтобы реализовать любую синтаксическую единицу, сохраняя гибкость в определениях, и сохраняя код чистым. Но у этого есть один минус – т.к. все обозначается как Node, Go не сможет нас предупредить если мы будем пытаться записать неверную синтаксическую структуру в другую, неподходящую для этого синтаксическую структуру, вследствие чего программисту нужно быть внимательным при задании правил и присваиваний во время анализа.

Например, так будет выглядеть структура корня дерева программы:

type ProgramNode struct {

Package string `json:"package"`

Body []Node `json:"body"`

Meta MetaData `json:"meta"`

}

Где Package – это название файла или модуля, Body – это список глобальных выражений, Meta – структура, необходимая для семантического анализа. Подобным образом мы определяем все ключевые синтаксические структуры, сохраняя в них только важную информацию. Дерево, в котором элементы использованию только ключевой информации называют абстрактным синтаксическим деревом.

Рассмотрим пример построения дерева на малом входном наборе данных “@const a:int = 12;”. Вывод представлен в формате json:

{

"body": [

{

"dataType": {

"meta": null,

"type": "symbolNode",

"value": {

"Column": 9,

"Line": 1,

"Type": 4,

"Value": "int"

}

},

"meta": null,

"name": {

"meta": null,

"type": "symbolNode",

"value": {

"Column": 7,

"Line": 1,

"Type": 4,

"Value": "a"

}

},

"type": "constNode",

"value": {

"meta": null,

"type": "integerNode",

"value": {

"Column": 15,

"Line": 1,

"Type": 2,

"Value": "12"

}

}

}

],

"meta": null,

"package": "",

"type": "programNode"

}

Все начинается в функции Parse(), которая создает корень дерева, структуру ProgramNode, и с помощью метода parseGlobal() обрабатывает глобальные синтаксические формы, заданный грамматикой:

func (p \*Parser) Parse() Node {

var programNode ProgramNode

for !p.isEOF() {

result := p.parseGlobal()

if result == nil {

return nil

}

programNode.Body = append(programNode.Body, result)

}

return programNode

}

Внутри parseGlobal() находится простое ветвление, которое на основе текущего токена определяет (можно сказать предугадывает), какая синтаксическая структура последует дальше, в нашем случае токен равен ‘@const’ и вследствии будет вызван метод parseConst():

func (p \*Parser) parseGlobal() Node {

tok := p.peek(0)

switch tok.Type {

case tokens.TokenConst:

return p.parseConst()

case tokens.TokenVar:

return p.parseVar()

case tokens.TokenFn:

return p.parseFunction()

default:

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want GLOBAL", tok.String()))

return nil

}

}

Метод parseConst() представляет из себя последовательность проверок, типичную для всех методов грамматик. Сначало он проверяет что нынешний токен это “@const” затем что это SYMBOL и так далее:

func (p \*Parser) parseConst() Node {

var constNode ConstNode

if p.peek(0).Type != tokens.TokenConst {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want 'const'", p.peek(0).String()))

return nil

}

p.next()

if p.peek(0).Type != tokens.TokenSymbol {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want SYMBOL", p.peek(0).String()))

return nil

}

constNode.Name = p.parseAtom()

if p.peek(0).Type != tokens.TokenColon {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want ':'", p.peek(0).String()))

return nil

}

p.next()

res := p.parseDataType()

if res == nil {

return nil

}

constNode.DataType = res

if p.peek(0).Type != tokens.TokenAssign {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want '='", p.peek(0).String()))

return nil

}

p.next()

res = p.parseAtom()

if res == nil {

return nil

}

constNode.Value = res

if p.peek(0).Type != tokens.TokenSemicolon {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want ';'", p.peek(0).String()))

return nil

}

p.next()

return constNode

}

Проверим работоспособность парсера с помощью некоторого текста программы, использующего в себе все синтаксические структуры:

@const a:int = 1;

@var b:int = (add 2 a);

@fn main:void() {

@const a:int = 1;

@set b = a;

@while (neq b 10) {

@set b = (add b 1);

}

@if (neg b 10) {

(printf "%d" b)

} else {

(printf "%d" (sub b 3))

}

}

Из этого примера можно выделить два основных алгоритма:

1. Алгоритм, который занимается распознаванием определенной грамматики, например функции parseConst(), Parse() и им подобные;

2. Алгоритм, который определяет, какая структура должна и будет следовать далее, основываясь на нынешнем состоянии парсера, например функции parseGlobal(), parseLocal(), parseAtom() и другие.

Алгоритм распознавания можно описать так:

НАЧАЛО

1. Создать пустой объект для данной грамматики.

2. Проверить, шаг за шагом, соответствие последовательности токенов к структуре грамматики, извлекая необходимые данные с помощью функций определения грамматики.

3. Вернуть полученный объект, либо выдать ошибку о несоответствии токенов и структуры грамматического правила.

КОНЕЦ

Алгоритм определения можно описать так:

НАЧАЛО

1. Выбрать текущей токен из списка токенов.

2. На основе типа токена, выбрать необходимую функцию распознавания, вернув ее результат выполнения.

3. Иначе вернуть ошибку о неожиданном токене.

КОНЕЦ

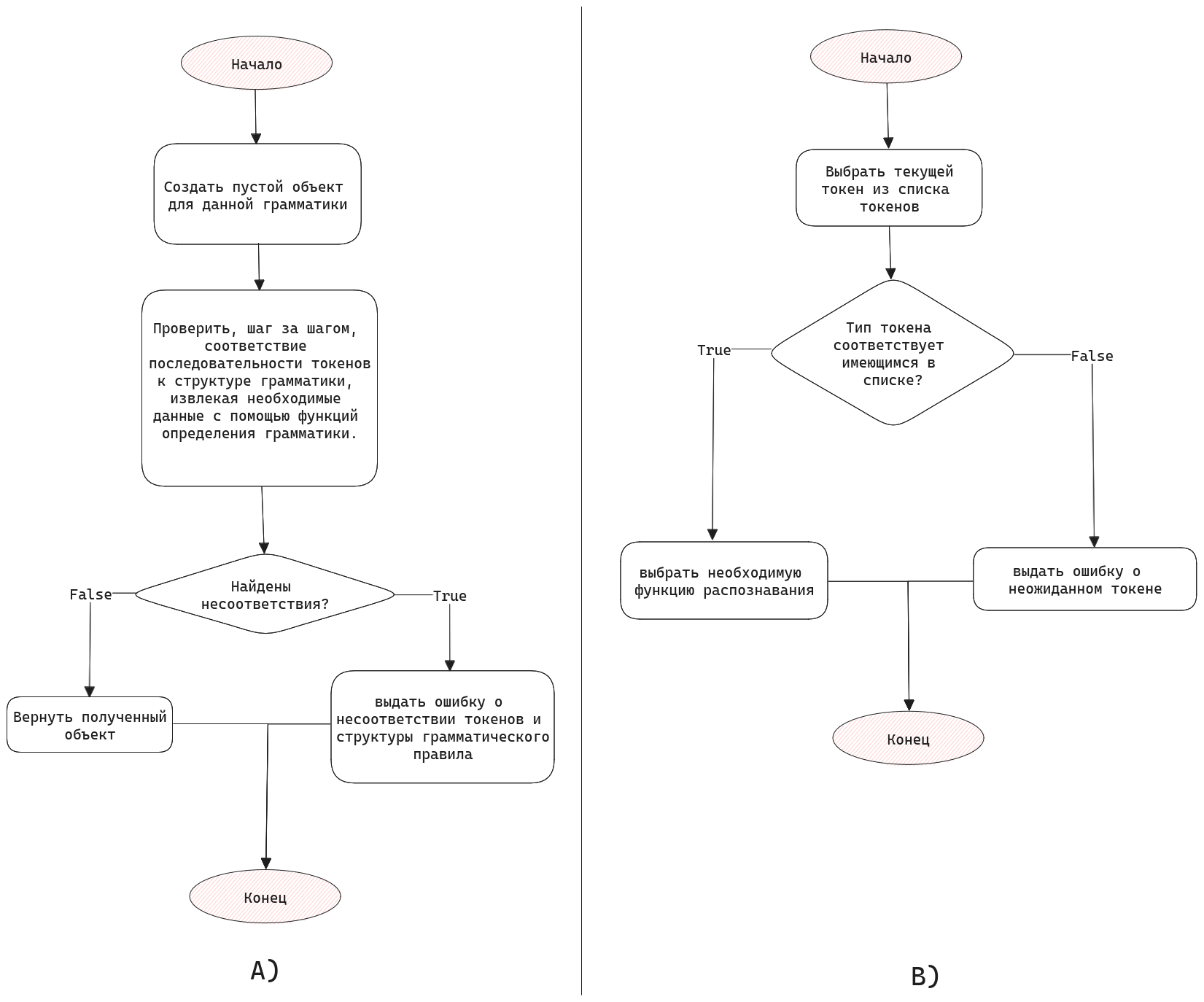


Рис. 3.3 Блок схемы для: A) Алгоритм распознавания; B) Алгоритм определения.

Результаты Unit-тестирования для каждой функции распознавания:

=== RUN TestParser

--- PASS: TestParser (0.00s)

=== RUN TestParseConst

--- PASS: TestParseConst (0.00s)

=== RUN TestParseVar

--- PASS: TestParseVar (0.00s)

=== RUN TestParseSet

--- PASS: TestParseSet (0.00s)

=== RUN TestParseIf

--- PASS: TestParseIf (0.00s)

=== RUN TestParseWhile

--- PASS: TestParseWhile (0.00s)

=== RUN TestParseFn

--- PASS: TestParseFn (0.00s)

=== RUN TestParseExpression

--- PASS: TestParseExpression (0.00s)

PASS

ok gl/core/frontend/parser/v2 0.003s

# 

# Заключение

В ходе данной данной работы были изучены: структура компилятора, устройство лексического анализатора, устройство синтаксического анализатора. Разработаны собственные выражения и грамматика для небольшого языка программирования, и на их основе созданы самописные лексический и синтаксический анализаторы. Все задачи нира выполнены.

## 

# 

# Список использованной литературы

1. Engineering of compiler: 2nd ed. *Keith D. Cooper, Linda Torczon*. Изд-во Morgan Kaufmann, Elsevier, 2011.- 824 с.
2. Programming language pragmatics: 4th ed. *Michael Scott.* Изд-во Morgan Kaufmann, 2015.- 992 с.
3. Modern Compiler Design: 2nd ed. *Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J. H. Jacobs, Koen Langendoen*. Изд-во Springer, 2012.- 843 с.
4. Writing Compilers and Interpreters: A Software Engineering Approach: 3rd ed. Ronald Mak. Изд-во Wiley, 2009.- 864 с.
5. [Электронный ресурс] The Go Programming Language

URL: <https://go.dev/> (Дата обращения 19.12.2022)

1. [Электронный ресурс] Bison - GNU Project - Free Software Foundation

URL: <https://www.gnu.org/software/bison/> (Дата обращения 23.11.2022)

1. [Электронный ресурс] ANTLR

URL: <https://www.antlr.org/> (Дата обращения 23.11.2022)

1. [Электронный ресурс] A parser library for Go - GitHub

URL: <https://github.com/alecthomas/participle> (Дата обращения 15.02.2023)

1. [Электронный ресурс] Should I use a parser generator or should I roll my own custom lexer and parser code? - Stackexchange

URL:<https://softwareengineering.stackexchange.com/questions/17824/should-i-use-a-parser-generator-or-should-i-roll-my-own-custom-lexer-and-parser> (Дата обращения 04.11.2022)

# Приложение А. Текст программы “Лексер”.

package frontend

import (

"fmt"

)

type Lexer struct {

input string

tokens []Token

errors []error

pos int

line int

column int

}

func NewLexer(input string) \*Lexer {

return &Lexer{

input: input,

}

}

func (l \*Lexer) Lex() \*[]Token {

for !l.isEOF() {

ch := l.peek(0)

switch ch {

case '\n':

l.next()

l.line++

l.column = 0

case '\r':

l.next()

case ' ', '\t':

l.next()

case '@':

tok := NewToken(TokenIllegal, "", l.line, l.column)

startPos := l.pos

l.next()

for {

ch = l.peek(0)

if !(ch >= 'a' && ch <= 'z') {

break

}

l.next()

}

tok.Value += l.input[startPos:l.pos]

tt, ok := IsKeyword(tok.Value)

if !ok {

l.errors = append(l.errors, fmt.Errorf("unknown keyword %s", tok.String()))

}

tok.Type = tt

l.tokens = append(l.tokens, \*tok)

case '(':

l.tokens = append(l.tokens, \*NewToken(TokenLeftParen, string(l.next()), l.line, l.column-1))

case ')':

l.tokens = append(l.tokens, \*NewToken(TokenRightParen, string(l.next()), l.line, l.column-1))

case '{':

l.tokens = append(l.tokens, \*NewToken(TokenLeftBrace, string(l.next()), l.line, l.column-1))

case '}':

l.tokens = append(l.tokens, \*NewToken(TokenRightBrace, string(l.next()), l.line, l.column-1))

case '=':

l.tokens = append(l.tokens, \*NewToken(TokenAssign, string(l.next()), l.line, l.column-1))

case '/':

l.tokens = append(l.tokens, \*NewToken(TokenSlash, string(l.next()), l.line, l.column-1))

case ':':

if l.peek(1) == ':' {

tok := NewToken(TokenDoubleColon, string(l.next()), l.line, l.column)

tok.Value += string(l.next())

l.tokens = append(l.tokens, \*tok)

} else {

l.tokens = append(l.tokens, \*NewToken(TokenColon, string(l.next()), l.line, l.column-1))

}

case ';':

l.tokens = append(l.tokens, \*NewToken(TokenSemicolon, string(l.next()), l.line, l.column-1))

case '"':

tok := NewToken(TokenString, "", l.line, l.column)

startPos := l.pos

l.next() // eat "

for l.isPrintable(l.peek(0)) && !l.isEOF() && l.peek(0) != '\n' {

l.next()

}

if l.isEOF() || l.peek(0) == '\n' {

l.errors = append(l.errors, fmt.Errorf("unterminated string at %d:%d", l.line, l.column))

continue

}

l.next() // eat "

tok.Value += l.input[startPos:l.pos]

l.tokens = append(l.tokens, \*tok)

default:

if l.isChar(ch) {

tok := NewToken(TokenIllegal, "", l.line, l.column)

startPos := l.pos

l.next()

for l.isChar(l.peek(0)) && !l.isEOF() {

l.next()

}

tok.Value += l.input[startPos:l.pos]

tt, ok := IsReserved(tok.Value)

if !ok {

tok.Type = TokenSymbol

} else {

tok.Type = tt

}

l.tokens = append(l.tokens, \*tok)

} else if l.isDigit(ch) {

tok := NewToken(TokenNumber, "", l.line, l.column)

startPos := l.pos

l.next()

for l.isDigit(l.peek(0)) && !l.isEOF() {

l.next()

}

if l.peek(0) == '.' {

l.next()

for l.isDigit(l.peek(0)) && !l.isEOF() {

l.next()

}

}

tok.Value += l.input[startPos:l.pos]

l.tokens = append(l.tokens, \*tok)

} else {

l.next()

l.errors = append(l.errors, fmt.Errorf("unexpected character '%c' at %d:%d", ch, l.line, l.column))

}

}

}

l.tokens = append(l.tokens, \*NewToken(TokenEOF, "", l.line, l.column))

return &l.tokens

}

func (l \*Lexer) peek(n int) byte {

if l.pos+n >= len(l.input) {

return 0

}

return l.input[l.pos+n]

}

func (l \*Lexer) next() byte {

if l.isEOF() {

return 0

}

c := l.input[l.pos]

l.pos++

l.column++

return c

}

func (l \*Lexer) isEOF() bool {

return l.pos >= len(l.input)

}

func (l \*Lexer) isChar(ch byte) bool {

return 'a' <= ch && ch <= 'z' || 'A' <= ch && ch <= 'Z'

}

func (l \*Lexer) isDigit(ch byte) bool {

return '0' <= ch && ch <= '9'

}

func (l \*Lexer) isPrintable(ch byte) bool {

return 0x20 <= ch && ch <= 0x7E && ch != '"'

}

func (l \*Lexer) Error() []error {

return l.errors

}

type TokenType uint8

const (

TokenIllegal TokenType = iota

TokenEOF

// literals

TokenNumber

TokenString

TokenSymbol

// delimeters

TokenLeftParen

TokenRightParen

TokenLeftBracket

TokenRightBracket

TokenLeftBrace

TokenRightBrace

// special symbols

TokenColon

TokenDoubleColon

TokenSemicolon

TokenAssign

TokenSlash

// keywords

TokenConst

TokenVar

TokenSet

TokenFn

TokenImport

Tokenlambda

TokenIf

TokenFor

TokenWhile

// reserved symbols

TokenElif

TokenElse

TokenFalse

TokenTrue

TokenAs

TokenTo

TokenFrom

TokenList

TokenMap

)

var tokenTypeMap = map[TokenType]string{

TokenEOF: "EOF",

TokenIllegal: "ILLEGAL",

TokenLeftParen: "(",

TokenRightParen: ")",

TokenLeftBracket: "[",

TokenRightBracket: "]",

TokenLeftBrace: "{",

TokenRightBrace: "}",

TokenColon: ":",

TokenDoubleColon: "::",

TokenSemicolon: ";",

TokenAssign: "=",

TokenSlash: "/",

TokenConst: "@const",

TokenVar: "@var",

TokenSet: "@set",

TokenFn: "@fn",

TokenImport: "@import",

Tokenlambda: "@lambda",

TokenIf: "@if",

TokenFor: "@for",

TokenWhile: "@while",

TokenElif: "elif",

TokenElse: "else",

TokenFalse: "false",

TokenTrue: "true",

TokenAs: "as",

TokenTo: "to",

TokenFrom: "from",

TokenList: "list",

TokenMap: "map",

TokenNumber: "NUMBER",

TokenString: "STRING",

TokenSymbol: "SYMBOL",

}

// keywords

var keywordMap = map[string]TokenType{

"@const": TokenConst,

"@var": TokenVar,

"@set": TokenSet,

"@fn": TokenFn,

"@import": TokenImport,

"@lambda": Tokenlambda,

"@if": TokenIf,

"@for": TokenFor,

"@while": TokenWhile,

}

func IsKeyword(val string) (TokenType, bool) {

tt, ok := keywordMap[val]

return tt, ok

}

// reserved symbols

var reservedMap = map[string]TokenType{

"elif": TokenElif,

"else": TokenElse,

"false": TokenFalse,

"true": TokenTrue,

"as": TokenAs,

"to": TokenTo,

"from": TokenFrom,

"list": TokenList,

"map": TokenMap,

}

func IsReserved(val string) (TokenType, bool) {

tt, ok := reservedMap[val]

return tt, ok

}

// Token

type Token struct {

Type TokenType

Value string

Line int

Column int

}

func (t Token) String() string {

return fmt.Sprintf("%d:%d(Type %s, Value %s)", t.Line, t.Column, tokenTypeMap[t.Type], t.Value)

}

func NewToken(tt TokenType, v string, lp, cp int) \*Token {

return &Token{

Type: tt,

Value: v,

Line: lp,

Column: cp,

}

}

# Приложение Б. Текст программы “Парсер”.

package frontend

import (

"encoding/json"

)

type Node interface {

json.Marshaler

node()

}

func (n ProgramNode) node() {}

func (n ConstNode) node() {}

func (n VarNode) node() {}

func (n SetNode) node() {}

func (n FunctionNode) node() {}

func (n WhileNode) node() {}

func (n IfNode) node() {}

func (n ExpressionNode) node() {}

func (n ParamNode) node() {}

func (n IntegerNode) node() {}

func (n FloatNode) node() {}

func (n StringNode) node() {}

func (n BooleanNode) node() {}

func (n SymbolNode) node() {}

type ProgramNode struct {

Package string `json:"package"`

Body []Node `json:"body"`

ScopeName string

}

func (progNode ProgramNode) MarshalJSON() ([]byte, error) {

type tmp ProgramNode

return toJson(tmp(progNode), "programNode")

}

type ConstNode struct {

Name Node `json:"name"`

DataType Node `json:"dataType"`

Value Node `json:"value"`

}

func (constNode ConstNode) MarshalJSON() ([]byte, error) {

type tmp ConstNode

return toJson(tmp(constNode), "constNode")

}

type VarNode struct {

Name Node `json:"name"`

DataType Node `json:"dataType"`

Value Node `json:"value"`

}

func (varNode VarNode) MarshalJSON() ([]byte, error) {

type tmp VarNode

return toJson(tmp(varNode), "varNode")

}

type SetNode struct {

Name Node `json:"name"`

Value Node `json:"value"`

}

func (setNode SetNode) MarshalJSON() ([]byte, error) {

type tmp SetNode

return toJson(tmp(setNode), "setNode")

}

type FunctionNode struct {

Name Node `json:"name"`

ReturnType Node `json:"returnType"`

Params []Node `json:"params"`

Body []Node `json:"body"`

ScopeName string

}

func (funcNode FunctionNode) MarshalJSON() ([]byte, error) {

type tmp FunctionNode

return toJson(tmp(funcNode), "functionNode")

}

type WhileNode struct {

ConditionExpressions Node `json:"conditionExpressions"`

ThenBody []Node `json:"thenBody"`

ScopeName string

}

func (whileNode WhileNode) MarshalJSON() ([]byte, error) {

type tmp WhileNode

return toJson(tmp(whileNode), "whileNode")

}

type IfNode struct {

ConditionExpressions Node `json:"conditionExpressions"`

ThenBody []Node `json:"thenBody"`

ElseBody []Node `json:"elseBody"`

ScopeName string

}

func (ifNode IfNode) MarshalJSON() ([]byte, error) {

type tmp IfNode

return toJson(tmp(ifNode), "ifNode")

}

type ParamNode struct {

Name Node `json:"name"`

DataType Node `json:"dataType"`

}

func (paramNode ParamNode) MarshalJSON() ([]byte, error) {

type tmp ParamNode

return toJson(tmp(paramNode), "paramNode")

}

type ExpressionNode struct {

Function Node `json:"function"`

Args []Node `json:"args"`

}

func (exprNode ExpressionNode) MarshalJSON() ([]byte, error) {

type tmp ExpressionNode

return toJson(tmp(exprNode), "expressionNode")

}

type IntegerNode struct {

Value Token `json:"value"`

}

func (intNode IntegerNode) MarshalJSON() ([]byte, error) {

type tmp IntegerNode

return toJson(tmp(intNode), "integerNode")

}

type FloatNode struct {

Value Token `json:"value"`

}

func (floatNode FloatNode) MarshalJSON() ([]byte, error) {

type tmp FloatNode

return toJson(tmp(floatNode), "floatNode")

}

type StringNode struct {

Value Token `json:"value"`

}

func (stringNode StringNode) MarshalJSON() ([]byte, error) {

type tmp StringNode

return toJson(tmp(stringNode), "stringNode")

}

type BooleanNode struct {

Value Token `json:"value"`

}

func (booleanNode BooleanNode) MarshalJSON() ([]byte, error) {

type tmp BooleanNode

return toJson(tmp(booleanNode), "booleanNode")

}

type SymbolNode struct {

Value Token `json:"value"`

}

func (symbolNode SymbolNode) MarshalJSON() ([]byte, error) {

type tmp SymbolNode

return toJson(tmp(symbolNode), "symbolNode")

}

// Helpers

func toJson(v any, tp string) ([]byte, error) {

res, \_ := json.Marshal(v)

var temp map[string]interface{}

json.Unmarshal(res, &temp)

temp["type"] = tp

return json.Marshal(temp)

}

// Parser

type Parser struct {

tokens []Token

pos int

errors []error

}

func NewParser(tokens []Token) \*Parser {

return &Parser{

tokens: tokens,

}

}

func (p \*Parser) Parse() Node {

var programNode ProgramNode

for !p.isEOF() {

result := p.parseGlobal()

if result == nil {

return nil

}

programNode.Body = append(programNode.Body, result)

}

return &programNode

}

// Parse functions

func (p \*Parser) parseGlobal() Node {

tok := p.peek(0)

switch tok.Type {

case TokenConst:

return p.parseConst()

case TokenVar:

return p.parseVar()

case TokenFn:

return p.parseFunction()

default:

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want GLOBAL", tok.String()))

return nil

}

}

func (p \*Parser) parseLocal() Node {

tok := p.peek(0)

switch tok.Type {

case TokenConst:

return p.parseConst()

case TokenVar:

return p.parseVar()

case TokenSet:

return p.parseSet()

case TokenWhile:

return p.parseWhile()

case TokenIf:

return p.parseIf()

case TokenLeftParen:

return p.parseExpression()

default:

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want LOCAL", tok.String()))

return nil

}

}

func (p \*Parser) parseAtom() Node {

tok := p.next()

switch tok.Type {

case TokenNumber:

if strings.Contains(tok.Value, ".") {

return &FloatNode{

Value: tok,

}

}

return &IntegerNode{

Value: tok,

}

case TokenString:

return &StringNode{

Value: tok,

}

case TokenTrue, TokenFalse:

return &BooleanNode{

Value: tok,

}

case TokenSymbol:

return &SymbolNode{

Value: tok,

}

default:

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want ATOM", tok.String()))

return nil

}

}

func (p \*Parser) parseDataType() Node {

tok := p.peek(0)

switch tok.Type {

case TokenSymbol:

return p.parseAtom()

case TokenList:

p.errors = append(p.errors, fmt.Errorf("unimplemented"))

return nil

case TokenMap:

p.errors = append(p.errors, fmt.Errorf("unimplemented"))

return nil

default:

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want DATA\_TYPE", tok.String()))

return nil

}

}

func (p \*Parser) parseExpressionArgument() Node {

tok := p.peek(0)

switch tok.Type {

case TokenSymbol, TokenNumber, TokenString, TokenTrue, TokenFalse:

return p.parseAtom()

case TokenLeftParen:

return p.parseExpression()

default:

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want EXPR\_ARG", tok.String()))

return nil

}

}

func (p \*Parser) parseIf() Node {

var ifNode IfNode

if !p.match(TokenIf) {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want 'if'", p.peek(0).String()))

return nil

}

p.next()

res := p.parseExpressionArgument()

if res == nil {

return nil

}

ifNode.ConditionExpressions = res

if !p.match(TokenLeftBrace) {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want '{'", p.peek(0).String()))

return nil

}

p.next()

for p.peek(0).Type != TokenRightBrace {

res := p.parseLocal()

if res == nil {

return nil

}

ifNode.ThenBody = append(ifNode.ThenBody, res)

}

if p.peek(0).Type != TokenRightBrace {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want '}'", p.peek(0).String()))

return nil

}

p.next()

if p.match(TokenElse) {

p.next()

if !p.match(TokenLeftBrace) {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want '{'", p.peek(0).String()))

return nil

}

p.next()

for p.peek(0).Type != TokenRightBrace {

res := p.parseLocal()

if res == nil {

return nil

}

ifNode.ElseBody = append(ifNode.ElseBody, res)

}

if p.peek(0).Type != TokenRightBrace {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want '}'", p.peek(0).String()))

return nil

}

p.next()

}

return &ifNode

}

func (p \*Parser) parseWhile() Node {

var whileNode WhileNode

if !p.match(TokenWhile) {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want 'while'", p.peek(0).String()))

return nil

}

p.next()

res := p.parseExpressionArgument()

if res == nil {

return nil

}

whileNode.ConditionExpressions = res

if !p.match(TokenLeftBrace) {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want '{'", p.peek(0).String()))

return nil

}

p.next()

for p.peek(0).Type != TokenRightBrace {

res := p.parseLocal()

if res == nil {

return nil

}

whileNode.ThenBody = append(whileNode.ThenBody, res)

}

if p.peek(0).Type != TokenRightBrace {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want '}'", p.peek(0).String()))

return nil

}

p.next()

return &whileNode

}

func (p \*Parser) parseExpression() Node {

var expressionNode ExpressionNode

if p.peek(0).Type != TokenLeftParen {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want '('", p.peek(0).String()))

return nil

}

p.next()

res := p.parseAtom()

if res == nil {

return nil

}

expressionNode.Function = res

for p.peek(0).Type != TokenRightParen {

res := p.parseExpressionArgument()

if res == nil {

return nil

}

expressionNode.Args = append(expressionNode.Args, res)

}

if p.peek(0).Type != TokenRightParen {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want ')'", p.peek(0).String()))

return nil

}

p.next()

return &expressionNode

}

func (p \*Parser) parseFunction() Node {

var functionNode FunctionNode

if p.peek(0).Type != TokenFn {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want 'fn'", p.peek(0).String()))

return nil

}

p.next()

if p.peek(0).Type != TokenSymbol {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want SYMBOL", p.peek(0).String()))

return nil

}

functionNode.Name = p.parseAtom()

if p.peek(0).Type != TokenColon {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want ':'", p.peek(0).String()))

return nil

}

p.next()

res := p.parseDataType()

if res == nil {

return nil

}

functionNode.ReturnType = res

if p.peek(0).Type != TokenLeftParen {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want '('", p.peek(0).String()))

return nil

}

p.next()

for p.peek(0).Type != TokenRightParen {

res := p.parseParameterNode()

if res == nil {

return nil

}

functionNode.Params = append(functionNode.Params, res)

}

if p.peek(0).Type != TokenRightParen {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want ')'", p.peek(0).String()))

return nil

}

p.next()

if p.peek(0).Type != TokenLeftBrace {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want '{'", p.peek(0).String()))

return nil

}

p.next()

for p.peek(0).Type != TokenRightBrace {

res := p.parseLocal()

if res == nil {

return nil

}

functionNode.Body = append(functionNode.Body, res)

}

if p.peek(0).Type != TokenRightBrace {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want '}'", p.peek(0).String()))

return nil

}

p.next()

return &functionNode

}

func (p \*Parser) parseParameterNode() Node {

var paramNode ParamNode

if p.peek(0).Type != TokenSymbol {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want SYMBOL", p.peek(0).String()))

return nil

}

paramNode.Name = p.parseAtom()

if p.peek(0).Type != TokenColon {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want ':'", p.peek(0).String()))

return nil

}

p.next()

res := p.parseDataType()

if res == nil {

return nil

}

paramNode.DataType = res

return &paramNode

}

func (p \*Parser) parseVar() Node {

var varNode VarNode

if p.peek(0).Type != TokenVar {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want 'var'", p.peek(0).String()))

return nil

}

p.next()

if p.peek(0).Type != TokenSymbol {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want SYMBOL", p.peek(0).String()))

return nil

}

varNode.Name = p.parseAtom()

if p.peek(0).Type != TokenColon {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want ':'", p.peek(0).String()))

return nil

}

p.next()

res := p.parseDataType()

if res == nil {

return nil

}

varNode.DataType = res

if p.peek(0).Type != TokenAssign {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want '='", p.peek(0).String()))

return nil

}

p.next()

res = p.parseExpressionArgument()

if res == nil {

return nil

}

varNode.Value = res

if p.peek(0).Type != TokenSemicolon {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want ';'", p.peek(0).String()))

return nil

}

p.next()

return &varNode

}

func (p \*Parser) parseSet() Node {

var setNode SetNode

if p.peek(0).Type != TokenSet {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want 'var'", p.peek(0).String()))

return nil

}

p.next()

if p.peek(0).Type != TokenSymbol {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want SYMBOL", p.peek(0).String()))

return nil

}

setNode.Name = p.parseAtom()

if p.peek(0).Type != TokenAssign {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want '='", p.peek(0).String()))

return nil

}

p.next()

res := p.parseExpressionArgument()

if res == nil {

return nil

}

setNode.Value = res

if p.peek(0).Type != TokenSemicolon {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want ';'", p.peek(0).String()))

return nil

}

p.next()

return &setNode

}

func (p \*Parser) parseConst() Node {

var constNode ConstNode

if p.peek(0).Type != TokenConst {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want 'const'", p.peek(0).String()))

return nil

}

p.next()

if p.peek(0).Type != TokenSymbol {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want SYMBOL", p.peek(0).String()))

return nil

}

constNode.Name = p.parseAtom()

if p.peek(0).Type != TokenColon {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want ':'", p.peek(0).String()))

return nil

}

p.next()

res := p.parseDataType()

if res == nil {

return nil

}

constNode.DataType = res

if p.peek(0).Type != TokenAssign {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want '='", p.peek(0).String()))

return nil

}

p.next()

res = p.parseAtom()

if res == nil {

return nil

}

constNode.Value = res

if p.peek(0).Type != TokenSemicolon {

p.errors = append(p.errors, fmt.Errorf("unexpected token %s, want ';'", p.peek(0).String()))

return nil

}

p.next()

return &constNode

}

// parser utilities

func (p \*Parser) match(expected TokenType) bool {

return p.peek(0).Type == expected

}

func (p \*Parser) next() Token {

tok := p.peek(0)

p.pos += 1

return tok

}

func (p \*Parser) peek(n int) Token {

if p.pos+n >= len(p.tokens) {

return p.tokens[len(p.tokens)-1]

}

return p.tokens[p.pos+n]

}

func (p \*Parser) isEOF() bool {

return p.match(TokenEOF)

}

func (p \*Parser) Errors() []error {

return p.errors

}