**Лексичний аналіз методом діаграми станів**

1. **Початкові дані для розробки лексичного аналізатора**
   1. **Завдання**

Варіант №19

**Арифметика**: цілі та дійсні числа, основні чотири арифметичні операції (додавання, віднімання, ділення та множення), піднесення до степеня(правоасоціативна операція), дужки

**Особливості**: експоненційна форма дійсного числа

**Інструкція повторення**: do <список операторів> while <логічний вираз> **Інструкція розгалуження:** if (<відношення>) {<сп. операторів>}

* 1. **Граматика мови**

Program = program Ident '{' DeclarList DoSection '}'.

DeclarList = '{' Declaration { Declaration } '}'.

Declaration = Type IdentList ';'.

IdentList = Ident{',' Ident }.

Type = int | float | boolean.

DoSection = '{' StatementList '}'.

StatementList = Statement {Statement}.

Statement = Assign | Inp | Out | ForStatement | IfStatement | LabelStatement | WhileStatement.

Assign = Ident '=' Expr ';'.

Expr = ArithmExpr | BooleanExpr.

BooleanExpr = Expr RelOp Expr | true | false.

ArithmExpr = [Sign] Term {AddOp Term}.

Term = Chunk {MultOp Chunk}.

Chunk = Factor {PowerOp Factor}.

Factor = Ident | Const | '(' Expr ')'.

Inp = input '(' IdenttList ')' ';'.

Out = print '(' IdenttList ')' ';'.

IfStatement = if '(' BooleanExpr ')' DoSection.

WhileStatement = do DoSection while '(' BooleanExpr ')'.

Const = IntNumber | FloatNumber | BoolConst.

IntNumber = [Sign] UnsignedInt.

FloatNumber = [Sign] UnsignedFloat.

Sign = '+' | '-'.

UnsignedInt = Digit {Digit}.

UnsignedFloat = UnsignedInt '.' UnsignedInt

| UnsignedInt '.' UnsignedInt 'E' IntNumber.

BoolConst = true | false.

Ident = Letter { Letter | Digit }.

Letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'.

Digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.

SpecialSign = '.' | ',' | ':' | ';' | '(' | ')' | '{' | '}' | '=' | '+' | '-' | '\*' | '/' | '^' | '<' | '>' | WhiteSpace | EndOfLine.

WhiteSpace = ' ' | '\t'.

EndOfLine = '\n' | '\r' | '\n\r'.

SpecSymbols = ArithOp | RelOp | BracketsOp | AssignOp | Punct.

ArithOp = AddOp | MultOp | PowerOp.

AddOp = '+' | '-'.

MultOp = '\*' | '/'.

PowerOp = '^'.

RelOp = '==' | '<=' | '<' | '>' | '>=' | '!='.

BracketsOp = '(' | ')'.

AssignOp = '='.

Punct = '.' | ',' | ':' | ';'.

* 1. **Таблиця лексем**

|  |  |  |  |
| --- | --- | --- | --- |
| Код | Приклади лексем | Токен | Неформальний опис |
| 1 | a, x1, z12f | іdent | ідентифікатор |
| 2 | 123, 0, 521 | іntnum | ціле без знака |
| 3 | 34.76 | floatnum | дійсне без знака |
| 4 | 1.2E03, 4.2E-01 | expnum | експоненційна форма дійсного числа |
| 5 | true, false | boolval | логічне значення |
| 6 | program | keyword | символ program |
| 7 | int | keyword | символ int |
| 8 | float | keyword | символ float |
| 9 | boolean | keyword | символ boolean |
| 10 | for | keyword | символ for |
| 11 | rof | keyword | символ rof |
| 12 | by | keyword | символ by |
| 13 | to | keyword | символ to |
| 14 | do | keyword | символ do |
| 15 | if | keyword | символ if |
| 16 | then | keyword | символ then |
| 17 | input | keyword | символ read |
| 18 | print | keyword | символ wrіte |
| 19 | = | assіgn\_op | символ = |
| 20 | + | add\_op | символ + |
| 21 | - | add\_op | символ - |
| 22 | \* | mult\_op | символ \* |
| 23 | / | mult\_op | символ / |
| 24 | ^ | power\_op | символ ^ |
| 25 | < | rel\_op | символ < |
| 26 | <= | rel\_op | символ <= |
| 27 | == | rel\_op | символ == |
| 28 | > | rel\_op | символ > |
| 29 | >= | rel\_op | символ >= |
| 30 | != | rel\_op | символ != |
| 31 | ( | par\_op | символ ( |
| 32 | ) | par\_op | символ ) |
| 33 | { | par\_op | символ { |
| 34 | } | par\_op | символ } |
| 35 | . | punct | символ . |
| 36 | , | punct | символ , |
| 37 | : | punct | символ : |
| 38 | ; | punct | символ ; |
| 39 | \32,\t | ws | пробільні символи |
| 40 | \n, \r, \r\n | ls | кінець рядка |

Табл. 1: Таблиця лексем мови LUMINAX

1. **Розробка лексичного аналізатора**
   1. **Побудова діаграми станів**
      1. **Класи символів**

Оберемо класи символів: Letter та Digit для символів, позначених однойменними нетерміналами в граматиці, клас punct – для символів ‘.’, ‘,’, ‘:’, ‘;’, клас ws – для пробільних символів, клас nl – для символу нового рядка, клас other – для символів, що не належать до поточної лексеми.

* + 1. **Діаграма станів (графічне представлення)**

Лексеми мови, представлені у Табл. 1. і такі, що містять більше одного символа, можуть бути розпізнані детермінованими автоматами, див Рис. 1.

На Рис. 1. представлена узагальнена для мови CIIPKA діаграма станів, отримана шляхом об’єднання стартових станів часткових діаграм та доповнення кожного не заключного стану переходом за символом класу other у певний заключний стан з міткою ERROR. На діаграмі представлені також переходи для односимвольних лексем – пробільних символів, символу нового рядка, арифметичних операторів та дужок.

* + 1. **Діаграма станів (символьне представлення)**

Діаграма станів як детермінований скінченний автомат:

Множина станів:

Алфавіт:

∑ = {a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,0,1,2,3,4,5,6,7,8,9, . , , , : , ; , = , + , - , \* , / , ^ , ( , ) , { , } , \n , \t, <, >, !=, ==, <=, >=, \_}

Функція переходів:

δ(0, Letter) = 1; δ(1, Letter) = 1; δ(1, Digit) = 1; δ(1, other) = 3;

δ(0, nl) = 5;

δ(0, sign) = 4;

δ(0, ws) = 0; δ(0, other) = 101;

δ(0, “>”) = 6; δ(0, “=”) = 6; δ(0, “<”) = 6; δ(0, “!”) = 9; δ(6, “=”) = 8; δ(9, “=”) = 8; δ(6, other) = 7; δ(0, other) = 102;

δ(0, Digit) = 10; δ(10, Digit) = 10; δ(10, other) = 12;

δ(10, “.”) = 11; δ(11, other) = 103; δ(11, Digit) = 14; δ(14, Digit) = 14; δ(14, other) = 12;

δ(14, “E”) = 15; δ(15, other) = 104; δ(15, “-”) = 17; δ(17, other) = 104; δ(15, Digit) = 13; δ(17, Digit) = 13; δ(13, Digit) = 13; δ(13, other) = 18;

Стартовий стан: q0 = 0

Множина заключних станів:

F = { 3,4,5,7,8,12,16,18,101,102,103,104}

Заключні стани, що потребують додаткової обробки:

F = {3,7,12,16,18}

FERROR = {101, 102, 103, 104}

* 1. **Cемантичні процедури**

Перехід до заключного стану діаграми передбачає виконання певних дій, які знаходяться за межами формалізму скінченних автоматів і які називають семантичними процедурами. Перелік семантичних процедур для мови LUMINAX наведено у Табл. 2.

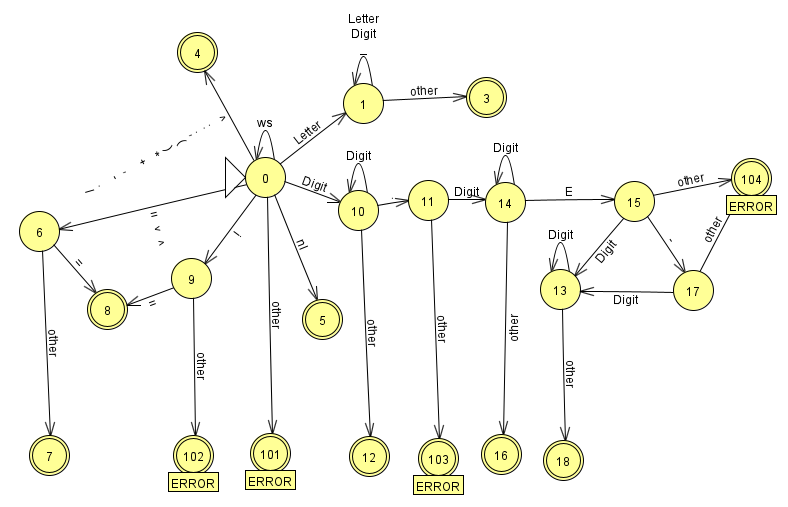


Рис. 1. Діаграма станів для розпізнавання довільних лексем мови LUMINAX

|  |  |  |
| --- | --- | --- |
| Стан | Токен | Семантичні процедури |
| 3 | ident, keyword, boolconst | Якщо ident – обробити таблицю ідентифікаторів  Занести лексему в таблицю розбору  Повернути необроблений символ у вхідний потік  Перейти у стартовий стан |
| 12, 16, 18 | int, float, exp | Обробити таблицю констант  Занести лексему в таблицю розбору  Повернути необроблений символ у вхідний потік  Перейти у стартовий стан |
| 5 | nl | Збільшити лічильник рядків на 1  Перейти у стартовий стан |
| 4 | add\_op, mult\_op, power\_op, punct | Занести лексему в таблицю розбору  Перейти у стартовий стан |
| 8 | rel\_op |
| 7 | assign\_op, rel\_op | Занести лексему в таблицю розбору  Повернути необроблений символ у вхідний потік  Перейти у стартовий стан |

Табл. 2. Семантичні процедури для мови LUMINAX

1. **Програмна реалізація лексичного аналізатора**
   1. **Діаграма станів і таблиця лексем у коді**

Лексичний аналізатор реалізується мовою C# на платформі .NET Core. Усі необхідні таблиці реалізуються як словники (Dictionary) мови C#.

* + 1. **Визначення класу вхідного символа**

public static class SymbolClass

{

public const string Digit = "Digit";

public const string DigitExample = "0123456789";

public const string Letter = "Letter";

public const string LetterExample = "abcdefghijklmnopqrstuvwxyz";

public const string WhiteSpaces = "ws";

public const string WhiteSpacesExample = " \t";

public const string NewLine = "nl";

public const string NewLineExample = "\n";

public const string OtherExample = "\'+-\*/^=(){}:E<>!,;.\"";

}

public static class SymbolAnalyzer

{

public static string GetClassOfSymbol(char charSymbol)

{

var symbol = charSymbol.ToString();

return symbol switch

{

{ } s when SymbolClass.LetterExample.Contains(s) => SymbolClass.Letter,

{ } s when SymbolClass.DigitExample.Contains(s) => SymbolClass.Digit,

{ } s when SymbolClass.WhiteSpacesExample.Contains(s) => SymbolClass.WhiteSpaces,

{ } s when SymbolClass.NewLineExample.Contains(s) => SymbolClass.NewLine,

{ } s when SymbolClass.OtherExample.Contains(s) => s,

\_ => "symbol doesn't belongs to alphabet"

};

}

}

* + 1. **Діаграма станів у коді**

Функція переходів δ представлена у формі словника (Dictionay) мови C#:

public static class States

{

public static Dictionary<StateTransition, int> StateTransitionsDictionary

= new() {

{ new StateTransition(0, "Letter"), 1},

{ new StateTransition(0, "nl"), 5},

{ new StateTransition(0, "ws"), 0},

{ new StateTransition(0, "+"), 4},

{ new StateTransition(0, "-"), 4},

{ new StateTransition(0, "\*"), 4},

{ new StateTransition(0, "/"), 4},

{ new StateTransition(0, "^"), 4},

{ new StateTransition(0, "("), 4},

{ new StateTransition(0, ")"), 4},

{ new StateTransition(0, "{"), 4},

{ new StateTransition(0, "}"), 4},

{ new StateTransition(0, ";"), 4},

{ new StateTransition(0, ","), 4},

{ new StateTransition(0, ":"), 4},

{ new StateTransition(0, "other"), 101},

{ new StateTransition(0, "="), 6},

{ new StateTransition(0, "<"), 6},

{ new StateTransition(0, ">"), 6},

{ new StateTransition(0, "!"), 9},

{ new StateTransition(0, "Digit"), 10},

{ new StateTransition(1, "Letter"), 1},

{ new StateTransition(1, "Digit"), 1},

{ new StateTransition(1, "other"), 3},

{ new StateTransition(3, "Digit"), 3},

{ new StateTransition(3, "other"), 4},

{ new StateTransition(6, "other"), 7},

{ new StateTransition(6, "="), 8},

{ new StateTransition(9, "="), 8},

{ new StateTransition(9, "other"), 102},

{ new StateTransition(10, "Digit"), 10},

{ new StateTransition(10, "other"), 12},

{ new StateTransition(10, "."), 11},

{ new StateTransition(11, "Digit"), 14},

{ new StateTransition(11, "other"), 103},

{ new StateTransition(14, "Digit"), 14},

{ new StateTransition(14, "other"), 16},

{ new StateTransition(14, "E"), 15},

{ new StateTransition(15, "-"), 17},

{ new StateTransition(17, "Digit"), 13},

{ new StateTransition(17, "other"), 104},

{ new StateTransition(15, "Digit"), 13},

{ new StateTransition(13, "Digit"), 13},

{ new StateTransition(13, "other"), 18},

};

public static List<int> FinalStates = new() { 3, 4, 5, 7, 8, 12, 16, 18, 101, 102, 103, 104 };

public static List<int> StatesToProcess = new() { 3, 4, 7, 12, 16 };

public static List<int> ErrorStates = new() { 101, 102, 103, 104 };

public const int InitState = 0;

}

* + 1. **Таблиця символів мови у коді**

Таблиця символів мови реалізована у формі двох словників – LanguageTokens та OtherTokens. Перший містить інформацію про токени, що однозначно представлені переліком лексем, другий – про ідентифікатори та константи.

public class Tokens

{

public static Dictionary<string, string> LanguageTokens = new()

{

{"program", "keyword"},

{"int", "keyword"},

{"float", "keyword"},

{"boolean", "keyword"},

{"while", "keyword"},

{"do", "keyword"},

{"if", "keyword"},

{"print", "keyword"},

{"input", "keyword"},

{"=", "assign\_op"},

{"<=", "rel\_op"},

{">=", "rel\_op"},

{"<", "rel\_op"},

{">", "rel\_op"},

{"==", "rel\_op"},

{"!=", "rel\_op"},

{".", "punct"},

{",", "punct"},

{":", "punct"},

{";", "punct"},

{"E", "punct"},

{" ", "ws"},

{"\t", "ws"},

{"\n", "nl"},

{"\r\n", "nl"},

{"-", "add\_op"},

{"+", "add\_op"},

{"\*", "mult\_op"},

{"/", "mult\_op"},

{"^", "pow\_op"},

{"(", "par\_op"},

{")", "par\_op"},

{"{", "par\_op"},

{"}", "par\_op"},

{"true", "boolval"},

{"false", "boolval"}

};

public static Dictionary<int, string> OtherTokens = new()

{

{ 3, "ident" },

{ 12, "int" },

{ 16, "float" },

{ 18, "exp"}

};

}

* 1. **Вихід лексичного аналізатора**
     1. **Резюме про роботу лексичного аналізатора**

При кожній ітерації перевірки діаграми станів викликається функція

яка аварійно зупиняє роботу програми.

private void CheckForErrors(int currentState, char symbol)

{

switch (currentState)

{

case 101:

throw new Exception($"101 Lexer: in line {LineOfCode} unexpected symbol '{symbol}'");

case 102:

throw new Exception($"102 Lexer: in line {LineOfCode} '=' was expected, received - '{symbol}'");

case 103:

throw new Exception($"103 Lexer: in line {LineOfCode} Digit was expected, received - '{symbol}'");

case 104:

throw new Exception($"104 Lexer: in line {LineOfCode} '-' or Digit were expected, received - '{symbol}'");

}

}

Функці, що викликається для аналізу рядку коду обернута в try catch

try

{

foreach (var lineOfText in TextReader.GetLineOfText(filePath))

{

lexicalAnalyzer.Analyze(lineOfText, ref currentState);

}

Console.WriteLine("Lexer: Lexical analyzer was successfully completed");

}

catch (Exception e)

{

var message = e.Message;

var errorCode = message.Substring(0, 4);

var mainMessage = message.Substring(2);

Console.WriteLine(mainMessage);

Console.WriteLine($"Lexer: Analysis failed with status {errorCode}");

}

* + 1. **Таблиця симолів програми (таблиця розбору):**

Таблиця розбору реалізована як словник SymbolsInformation у форматі:

public Dictionary<int, SymbolInformation> SymbolsInformation = new();

де ключем слугує номер запису в таблиці символів програми, а значенням є

public record SymbolInformation(int LineNumber, string Lexeme, string LexemeToken, int? Index); де

LineNumber – номер рядка вхідної програми;

Lexeme – лексема;

LexemeToken – токен лексеми;

Index – індекс ідентифікатора або константи у таблиці ідентифікаторів та констант відповідно; для інших лексем - null.

* + 1. **Таблиця ідентифікаторів**

Таблиця розбору реалізована як словник Ids у форматі:

public Dictionary<string, int> Ids = new();

де ключ – лехема, а значення - Id у таблиці

* + 1. **Таблиця констант**

Таблиця розбору реалізована як словник Ids у форматі:

public Dictionary<string, int> Ids = new();

де ключ – лехема, а значення - Id у таблиці

* 1. **Програмна реалізація основних функцій**
     1. **Функції верхнього рівня**

Програма починає свою роботу з порядкового зчитування вихідного файлу та поступовий його аналіз

foreach (var lineOfText in TextReader.GetLineOfText(filePath))

{

lexicalAnalyzer.Analyze(lineOfText, ref currentState);

}

public static class TextReader

{

public static IEnumerable<string> GetLineOfText(string filePath)

{

if (!File.Exists(filePath))

{

throw new FileNotFoundException("This file doesn't exist");

}

foreach (var line in File.ReadLines(filePath))

{

yield return line;

}

}

}

У класі lexicalAnalyzer є одна публічна функція в яку передається строка з кодом та поточний стан у діаграмі стану

public class LexicalAnalyzer

{

public int LineOfCode { get; set; } = 1;

public AnalysisInformation AnalysisInformation { get; } = new();

private int \_counter;

public void Analyze(string lineOfCode, ref int currentState)

{

var lexeme = string.Empty;

for (; \_counter <= lineOfCode.Length - 1; \_counter++)

{

var symbolClass = SymbolAnalyzer.GetClassOfSymbol(lineOfCode[\_counter]);

currentState = GetState(currentState, symbolClass);

CheckForErrors(currentState, lineOfCode[\_counter]);

ProcessStates(lineOfCode[\_counter], ref lexeme, ref currentState);

}

\_counter = 0;

LineOfCode++;

}

…

}

private void ProcessStates(char symbol, ref string lexeme, ref int currentState)

{

if (States.FinalStates.Contains(currentState) || States.StatesToProcess.Contains(currentState))

{

ProcessFinalState(ref lexeme, currentState, symbol);

currentState = States.InitState;

}

else if (currentState == States.InitState)

{

lexeme = string.Empty;

}

else

{

lexeme += symbol;

}

}

* + 1. **Переходи у діаграмів стану**

Функція GetState(int currentState, string symbolClass) перевіряє наявність у словнику StateTransitionsDictionary значення за ключем (currentState, symbolClass). Якщо таке значення знайдено, то воно і є наступним станом, якщо ж запису з таким ключем немає – то в словнику здійснюється пошук ключа (currentState, other).

private int GetState(int currentState, string symbolClass)

{

var state = -1;

if (States.StateTransitionsDictionary

.TryGetValue(new StateTransition(currentState, symbolClass), out var existingState))

{

state = existingState;

}

else if (States.StateTransitionsDictionary

.TryGetValue(new StateTransition(currentState, "other"), out var otherState))

{

state = otherState;

}

return state;

}

* + 1. **Реалізація сематичних процедур**

private void ProcessFinalState(ref string lexeme, int currentState, char symbol)

{

var token = GetToken(lexeme, currentState, symbol);

int? indexOfConstOrIndent = null;

if (token == "nl")

{

lexeme = $"\\n";

LineOfCode++;

}

else if (token is "keyword")

{

\_counter--;

}

else if (token == "ident")

{

if (!AnalysisInformation.Ids.ContainsKey(lexeme))

{

indexOfConstOrIndent = AnalysisInformation.Ids.Count + 1;

AnalysisInformation.Ids.Add(lexeme, (int)indexOfConstOrIndent);

}

\_counter--;

}

else if (new[] { "int", "float", "exp", "boolval" }.Contains(token))

{

if (!AnalysisInformation.Constants.ContainsKey(lexeme))

{

indexOfConstOrIndent = AnalysisInformation.Ids.Count + 1;

AnalysisInformation.Constants.Add(lexeme, (int)indexOfConstOrIndent);

}

\_counter--;

}

else if (new[] { "punct", "add\_op", "mult\_op", "pow\_op", "par\_op", "rel\_op" }.Contains(token))

{

lexeme += symbol;

}

var data = new SymbolInformation(LineOfCode, lexeme, token, indexOfConstOrIndent);

AnalysisInformation.SymbolsInformation

.Add(AnalysisInformation.SymbolsInformation.Count, data);

Console.WriteLine("");

lexeme = string.Empty;

}

private string GetToken(string lexeme, int state, char symbol)

{

var result = string.Empty;

if (Tokens.LanguageTokens.TryGetValue(lexeme, out var languageToken))

{

result = languageToken;

}

else if (Tokens.OtherTokens.TryGetValue(state, out var numberToken))

{

result = numberToken;

}

else if (Tokens.LanguageTokens.TryGetValue(lexeme + symbol, out var relToken))

{

result = relToken;

}

else if (Tokens.LanguageTokens.TryGetValue(symbol.ToString(), out var symbolToken))

{

result = symbolToken;

}

return result;

}

1. **Тестування роботи лексичного аналізатор**
   1. **Базовий приклад**

program myprogram {

{

int x,y,z, i;

float a,b;

boolean c;

}

{

a = x ^ y;

print(x, y);

input(x,y);

x = x + 3;

y = y \* 2;

x = x - 2;

y = y / 3;

a = x ^ y;

b = 1.3E02;

c = (5 <= 3);

i = 4;

do {

if (i / 2 != 0) {

print(i);

}

i = i - 1;

} while (i >= 0)

}

}

**Результат виконання**

1 program keyword

1 myprogram ident 1

1 { par\_op

2 { par\_op

3 int keyword

3 x ident 2

3 , punct

3 y ident 3

3 , punct

3 z ident 4

3 , punct

3 i ident 5

3 ; punct

4 float keyword

4 a ident 6

4 , punct

4 b ident 7

4 ; punct

5 boolean keyword

5 c ident 8

5 ; punct

6 } par\_op

7 { par\_op

8 a ident

8 = assign\_op

8 x ident

8 ^ pow\_op

8 y ident

8 ; punct

9 print keyword

9 ( par\_op

9 x ident

9 , punct

9 y ident

9 ) par\_op

9 ; punct

10 input keyword

10 ( par\_op

10 x ident

10 , punct

10 y ident

10 ) par\_op

10 ; punct

11 x ident

11 = assign\_op

11 x ident

11 + add\_op

11 3 int 9

11 ; punct

12 y ident

12 = assign\_op

12 y ident

12 \* mult\_op

12 2 int 9

12 ; punct

13 x ident

13 = assign\_op

13 x ident

13 - add\_op

13 2 int

13 ; punct

14 y ident

14 = assign\_op

14 y ident

14 / mult\_op

14 3 int

14 ; punct

15 a ident

15 = assign\_op

15 x ident

15 ^ pow\_op

15 y ident

15 ; punct

16 b ident

16 = assign\_op

16 1.3E02 exp 9

16 ; punct

17 c ident

17 = assign\_op

17 ( par\_op

17 5 int 9

17 <= rel\_op

17 3 int

17 ) par\_op

17 ; punct

18 i ident

18 = assign\_op

18 4 int 9

18 ; punct

19 do keyword

19 { par\_op

20 if keyword

20 ( par\_op

20 i ident

20 / mult\_op

20 2 int

20 != rel\_op

20 0 int 9

20 ) par\_op

20 { par\_op

21 print keyword

21 ( par\_op

21 i ident

21 ) par\_op

21 ; punct

22 } par\_op

23 i ident

23 = assign\_op

23 i ident

23 - add\_op

23 1 int 9

23 ; punct

24 } par\_op

24 while keyword

24 ( par\_op

24 i ident

24 >= rel\_op

24 0 int

24 ) par\_op

25 } par\_op

26 } par\_op