Dokumentacja do projektu z TKOM

Roman Moskalenko
18 maja 2020

Spis treści

1 Treść zadania

Interpretacja własnego języka z wbudowanym typem macierzy dwuwymiarowej.

2 Projekt wstępny

2.1 Opis ogólny

Fantastic Matrix Language (Fantastyczny język macierzowy, dalej FML) jest językiem skryptowym dynamicznie typowanym. Można go traktować jako uproszczone połączenie *Pythona* i *C*.

Wszystkie białe znaki (whitespace) są ignorowane [1] przez interpreter, dlatego nie zostaną uwzględnione na większości diagramów co pozwoli je uprościć. Także ignorowane są komentarze poprzedzone znakiem # (jednoliniowe).

Interpreter musi być w stanie obsługiwać różne typy źródeł: string, plik, strumień.

2.2 Słowa kluczowe

FML łącznie zawiera 14 słów kluczowych, które są przedstawione poniżej.

```
and div do else for fun if
in mod not or ret while
```

- Operatory logiczne: **and, not, or**
- Konstrukcje pętlowe: **do, for, in, while**
- Operatory arytmetyczne: **div, mod**
- Konstrukcje warunkowe: **else, if**
- Konstrukcje funkcji: **fun, ret**

[1] Wyjątkiem są znaki będące zawartością stringa(zapisane pomiędzy nawiasów "").

2.3 Identyfikatory i typy danych

2.3.1 Indentyfikator

Identyfikatorem jest zmienna, która może reprezentować jeden z trzech typów danych bądź funkcję.

FML oferuje 3 typy danych: **skalar**, **macierz** oraz **string**.

2.3.2 Skalar

Skalarem jest stała liczba. Możliwy jest zapis w notacji naukowej.

Przykłady:

- 0, 1, -12.3, 12.3e4, 12.3e-45.

2.3.3 String

String jest to łańcuch symboli poprzedzony i zakończony znakiem **"**. String nie może brać udziału w wyrażeniach arytmetycznych lub logicznych. Nie może być przechowywany w macierzach. Może być przekazywany jako argument w niektórych funkcjach, stosowany w przypisywaniach. Jest iterowalny.

Przykłady:

- "Hello world!`\n`"
- "Ala nie ma`\t` kota"

2.3.4 Macierz

Macierz jest tablicą dwuwymiarową, która może zawierać tylko skalary.

Znak **,** jest używany dla separacji kolejnych elementów w jednym wierszu macierzy. Znak **;** jest używany dla separacji kolejnych wierszów. Wiersz macierzy nie może być pusty, chyba, że cała macierz jest pusta. Liczba elementów w każdym wierszu macierzy musi być taka sama.

Przykłady:

- [ ]
- [1, 2, 3, 4, 5]
- [1, 2, 3; 4, 5, 6; 7, 8, 9]
- [1, 2, 3, 4;
  5, 6, 7, 8]

2.3.5 Indeksowanie

Odwołanie do elementów macierzy bądź stringa jest możliwe dzięki operatorowi [ ]. Do elementu macierzy można dostać się za pomocą dwóch indeksów odseparowanych przecinkiem: pierwszy specyfikuje wiersz macierzy, drugi – kolumnę. Podając zamiast indeksu znak **:** można odwołać się do wszystkich elementów wiersza/kolumny. W przypadku ujemnego indeksu zostanie zwrócony element licząc od końca.

Przykłady:

```
m = [1, 2, 3;
     4, 5, 6]

m[0, 1] # 2

m[:, 1] # [2, 5]

m[1,:] # [4, 5, 6]
```

W przypadku podania tylko jednego indeksu bez separatora zostanie zwrócony element z wypłaszczonej macierzy.

```
s = "Hello world!"

s[4] # o

m = [1, 2, 3, 4; 5, 6, 7, 8]

m[5] # 6
```

2.4 Wyrażenia i operatory

2.4.1 Operatory

Operatory są przedstawione w tabeli poniżej.

Priorytet	Operator	Opis
1	**	Potęgowanie
2	+ - not	Jednoargumentowy plus i minus Logiczne zaprzeczenie
3	* / mod div	Mnożenie, dzielenie oraz modulo Dzielenie całkowitoliczbowe
4	+ -	Dodawanie, odejmowanie
5	< <= > >=	Operatory $<$ oraz $\leqslant$ Operatory $>$ oraz $\geqslant$
6	== !=	Operatory $=$ oraz $\neq$
7	and	Logiczny iloczyn
8	or	Logiczna suma
9	=	Przypisanie

2.4.2 Wyrażenia

Wyrażenie może być pojedynczą stałą, zmienną lub też pewną operacją na zmiennych. Kolejność wykonywanych operacji jest zdefiniowana za pomocą priorytetów operatorów opisanych powyżej. Także mogą być stosowane nawiasy okrągłe: (i). Wyrażenie nie może być puste, nie może zawierać samych nawiasów.

2.4.3 Przypisanie wartości identyfikatorowi

Przypisanie jest szczególnym przypadkiem wyrażenia, jest dokonywane za pomocą operatora przypisania =.

2.5 Konstrukcje warunkowe, funkcje, pętle

2.5.1 Wyrażenia logiczne

Zanim przejść do konstrukcji warunkowych trzeba zdefiniować jaką wartość może przyjmować wyrażenie logiczne. FML nie ma specjalnego typu wartości logicznych.

Natomiast przyjęto takie założenia.

Wyrażenie logiczne jest **fałszywe** gdy jest równe jednej z poniższych wartości:

- Skalar zerowy **0**.
- Pusta macierz [ ].

W pozostałych przypadkach wyrażenie logiczne będzie **prawdziwe**.

Jeśli wyrażenie logiczne jest prawdziwe to jego wartość jest równa 1. W przypadku gdy wyrażenie logiczne nie jest prawdziwe wartość jego jest zerowa. Pozwala to na wykorzystanie w wyrażeniach operacji arytmetycznych w połączeniu z operacjami logicznymi.

Przykład:

```
a = 1 + (1 > 0); # a = 2
```

2.5.2 Konstrukcje warunkowe

Do konstrukcji warunkowych używane są słowa kluczowe **if** oraz **else**. Gramatyka języka umożliwia zagnieżdżanie tych konstrukcji.

Przykłady:

```
if (a<b) a=b;
else {
  a = a - 10;
  b = b * 2;
}

if (a)
  if (b) {
    b = c+d;
  }
  else
    a = c+d;

if (a) {
  if (b)
    b = b + 10;
} else {
  a = a + 10;
}
```

2.5.3 Funkcje

Deklaracja funkcji rozpoczyna się od słowa kluczowego **fun**. Po nim następuje identyfikator funkcji. Parametry funkcji należy opisać w nawiasach jako identyfikatory odseparowane przecinkiem. Nawiasy puste oznaczają brak parametrów. Na koniec jest ciało funkcji.

Przykład:

```
fun my_function(parameter1, parameter2)
  a = parameter1 + parameter2;
```

Aby zwrócić wartość przez funkcję należy użyć słowa kluczowego **ret**. Domyślną wartością zwracaną przez funkcje jest **0**.

Przykład:

```
fun my_function(parameter)
  ret parameter + 1;
```

Aby wywołać istniejącą funkcje należy podać jej identyfikator oraz argumenty wywołania w nawiasach okrągłych.

Przykład:

```
my_function();
my_function(1);
my_function(a, b);
```

Funkcje mogą być definiowane przez użytkownika tylko w globalnym zakresie programu (nie można definiować węwnatrz pętli, innych funkcji itd.)

2.5.3.1 Funkcje wbudowane

Do wbudowanych należa funkcje:

- abs() – zwraca wartość bezwzględną argumentu,
- len() – zwraca długość (liczbę elementów) objektu,
- max() – zwraca największą liczbę z podanej sekwencji,
- min() – zwraca najmniejszą liczbę z podanej sekwencji,
- print() – wypisuje na konsole podane argumenty,
- round() – zwraca zaokrąglony skalar,
- shape() – zwraca macierz zawierającą wymiary podanego argumentu,
- transp() – odwraca podaną macierz.

2.5.4 Pętle

W FML wyróżnia się 3 typy pętli: **for loop**, **while loop**, **do while loop**.

Pętle **for loop** przeznaczone do iterowania po macierzy lub stringu. Do tego używa się słowa kluczowego **in**, który rozwija iterowalny zbiór.

Przykład wyliczenia sumy wszystkich elementów macierzy:

```
m = [1, 2; 3.5, 4.5];
s = 0.0;
for (i in m) {
  s = s + i;
}
```

Pętla mogła również zostać zapisana w sposób następujący:

```
...
for (i in m)
  s = s + i;
```

W pętlach **while** instrukcje są wykonywane póki jest spełniony warunek. W przypadku **do while loop** zawartość pętli uruchomi się co najmniej raz niezależnie od tego czy spełniony warunek pętli. Składnia jest analogiczna do języka **C**.

Przykłady wypisania liczb od 1 do 10:

```
i = 1
while (i <= 10) {
  print(i);
  i = i + 1;
}

i = 1;
do {
  print(i);
  i = i + 1;
} while (i <= 10);
```

FML nie rozpoznaje słów kluczowych **break** i **continue**. Nie jest możliwe ich użycie wewnątrz konstrukcji pętlowych.

3 Dokumentacja końcowa

3.1 Zmiany w projekcie wstępnym

Dla rozszerzenia funkcjonalności, a w niektórych przypadkach dla uproszczenia implementacji, zostały wprowadzone następujące zmiany:

- Do tej pory *String* nie był częścią wyrażeń. Zmiana umożliwia wykorzystanie objektów *String* w wyrażeniach, a co za tym idzie w wywołaniach funkcji jako argument.
- Pusty *String* w wyrażeniach logicznych jest traktowany jako wartość fałszywa.
- Został wprowadzony obiekt *EmptyStatement*. Musi być zakończony średnikiem.
- Pozwolono na zagnieżdżone przypisywania.
- Zrezygnowano z ujemnych indeksów przy odwołaniu się do elementów macierzy.
- Zrezygnowano z opcji źródła danych ze strumienia.
- Zostały poprawione diagramy składni.

3.2 Opis struktury projektu

Struktura plików i folderów w projekcie.

```
.
├── Error.py
├── Source
│   ├── Source.py
│   └── Position.py
├── Lexer
│   ├── Lexer.py
│   └── Token.py
├── Parser
│   └── Parser.py
├── Objects
│   ├── Builtins.py
│   ├── Function.py
│   ├── Identifier.py
│   ├── Matrix.py
│   ├── Operators.py
│   ├── Program.py
│   ├── Scalar.py
│   ├── Statement.py
│   └── String.py
├── Interpreter
│   ├── Ast.py
│   ├── AstDumper.py
│   ├── TextTreeStructure.py
│   ├── Environment.py
│   └── Interpreter.py
├── tests
│   ├── lexer
│   ├── parser
│   └── interpreter
├── examples
└── fmli.py
```

3.2.1 Error.py

Plik zawiera kody błędów zdefioniowane dla FML oraz klasy błędów.

- Klasa *Error* - podstawowa klasa dziedzicząca po natywnej klasie Python'a *Exception*, co umożliwia rzucanie wyjątkiem za pomocą tej klasy

oraz jej pochodnych.

- Klasy *LexerError*, *ParserError*, *InterpreterError* - klasy dziedziczące po klasie Error.

3.2.2 Source

Folder Source zawiera dwa pliki: Source.py oraz Position.py.

Plik Source.py zawiera trzy klasy: *Source* oraz dziedziczące po niej *FileSource* oraz *StringSource* specjalnie dostosowane do obsługi odpowiednio plików oraz stringów jako danych wejściowych dla interpretera.

Zadaniem klasy *Source* jest przygotowanie źródła i zwracania z niego po jednym symbolu, a także śledzenie aktualnie przetwarzanej pozycji w źródle.

3.2.3 Lexer

Folder Lexer zawiera dwa pliki: Lexer.py oraz Token.py.

W pliku Token.py zdefiowane są klasy *TokenType* i *Token*.

TokenType trzyma w sobie wszytkie typy tokenów rozumiane przez Lexer, a także słowa kluczowe właściwe dla FML.

Klasa *Lexer* jest odpowiedzialna za przetworzenie źródła danych i wyprodukowanie obiektów Token. Obiekt tej klasy dostaje od obiektu klasy *Source* symbole i próbuje je dopasowywać do znanych typów tokenów.

3.2.4 Parser

Folder Parser zawiera pojedynczy plik Parser.py.

W tym pliku znajduje się jedynie klasa *Parser*, zadaniem której jest przetworzenie tokenów otrzymanych od obiekta Lexera i wytworzenie obiektów potrzebnych do stworzenia AST drzewa. Parser parsuje obiekty kierując się gramatyką FML, diagramy ilustrujące ją znajdują się na końcu dokumentacji.

3.2.5 Objects

W tym folderze znajdują się klasy, obiekty których są produkowane przez parser z wyjątkiem definicji funkcji wbudowanych (plik Builtins.py), które zostają dodane do AST dopiero na etapie interpretera. Obiekty, które zostają dodane do AST, dziedziczą po specjalnej klasie nazwanej *AST* bądź z pochodnych tej klasy.

- Function.py - Zawiera klasy *FunctionDefinition* oraz *FunctionCall*.

- Identifier.py - klasa definiująca obiekt identyfikatora bądź zmiennej w FML.

- Matrix.py - zawiera klasy *Matrix*, *MatrixRow*, *MatrixIndex*, *MatrixSubscripting*. Klasy *Matrix* oraz *MatrixRow* wykorzystywane do zapisu definicji macierzy, pozostałe dwie klasy obsługują wyłuskanie przy pomocy indeksów elementów macierzy.

- Operators.py - zawiera kluczowe klasy, z obiektów których zbudowane są wyrażenia w FML. *BinaryOperator* - obiekt trzyma operator oraz wartości po obu jego stronach, *Assingment* - obsługuje specyficzną sytuację operatora binarnego gdzie operatorem jest znak przypisania. *UnaryOperator* - dla zapisu jednoargumentowego operatora i jego argumentu po prawej stronie.

- Program.py - obiekt tej klasy trzyma obiekty najwyższego poziomu, czyli definicje funkcji oraz obiekty klasy *Statement* w zakresie globalnym. Interpreter zaczyna swoje działanie odwiedzając obiekt tej właśnie klasy.

- Scalar.py - klasa reprezentująca stałe liczbowe. FML operuje tylko na skalarach zmiennoprzecinkowych, które za potrzeby konwertowane na liczby całkowite.

- Statement.py - w tym pliku znajdują się definicje takich konstrukcji jak instrukcja warunkowa *IfStatement*, pętle *DoWhile*, *While* oraz *For in*. Także są tam klasy *EmtpyStatement*, *CompoundStatement* oraz *ReturnStatement*.

- String.py - klasa, która opakowuje zwykły string.

3.2.6 Interpreter

W tym folderze znajduje się wszystko co jest potrzebne do interpretacji przeparsowanych obiektów, a także wyświetlanie drzewa AST w postaci tekstowej.

- Ast.py - zawiera abstrakcyjna klasę *AST*, po której dziedziczą wszystie obiekty trafiające do AST, oraz klasę *NodeVisitor* - podstawę dla implementacji wzorcu Wizytora.

- TextTreeStructure.py - przeportowany z C++ dumper clang'a AST drzew na postać tekstową.

- AstDumper.py - klasa dziedziczy po *AST* oraz *TextTreeStructure*. Odwiedza obiekty drzewa i wypisuje je na konsole.
- Interpreter.py - klasa dziedzicząca po *NodeVisitor*. Odwiedza obiekty drzewa AST i interpretuje je. Dla odwiedzenia każdego obiektu wymagana jest metoda odpowiednia do konkretnej klasy, która wygląda tak: *visit_NazwaKlasy*. Także klasa interpretera zawiera wszystkie metody implementujące operacje oferowane przez FML.
- Environment.py - zawiera klasy *Environment*, *Scope* oraz *GlobalScope*. Obiekt klasy *Environment* oferuje interfejs dla obiektu interpretera. Environment ma pola zakresów globalnych (*GlobalScope*): global scope oraz outer scope, a także trzyma wskazanie na bieżący zakres (current scope). Oprócz tego ma stos, na które odkładane zostają zakresy sprzed wywołaniem funkcji aby móc je potem przywrócić po powrocie z funkcji. Także obiekt klasy *Environment* odpowiada za przygotowanie definicji funkcji wbudowanych.

3.2.7 tests

W celu sprawdzania poprawności działania zostały napisane testy dla trzech głównych części projektu: leksera, parsera i interpretera. Zarówno są testy pozytywne, jak i negatywne.

3.2.8 examples

Przygotowane zostały działające przykłady programów napisanych w FML.

3.3 Sposób wykorzystania

Aby uruchomić interpreter należy wykonać w konsoli polecenie

```
$ python fmli.py path/to/file
```

co zinterpretuje zawartość podanego pliku.

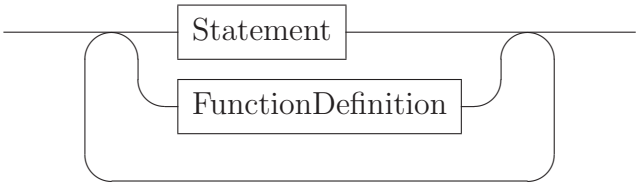
Aby wyświetlić drzewo AST w postaci tekstowej należy do polecenia dodać flagę **-d** lub **- -dump**.

```
$ python fmli.py path/to/file –d
```

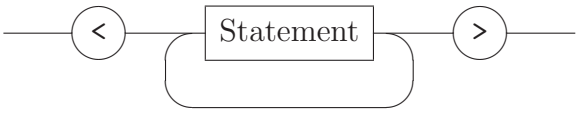
Minimalna wersja Python wymagana do pracy programu – 3.8.

4 Diagramy składniowe

Program

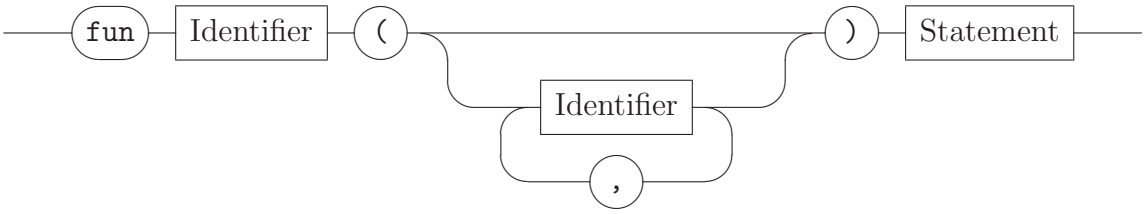


CompoundStatement



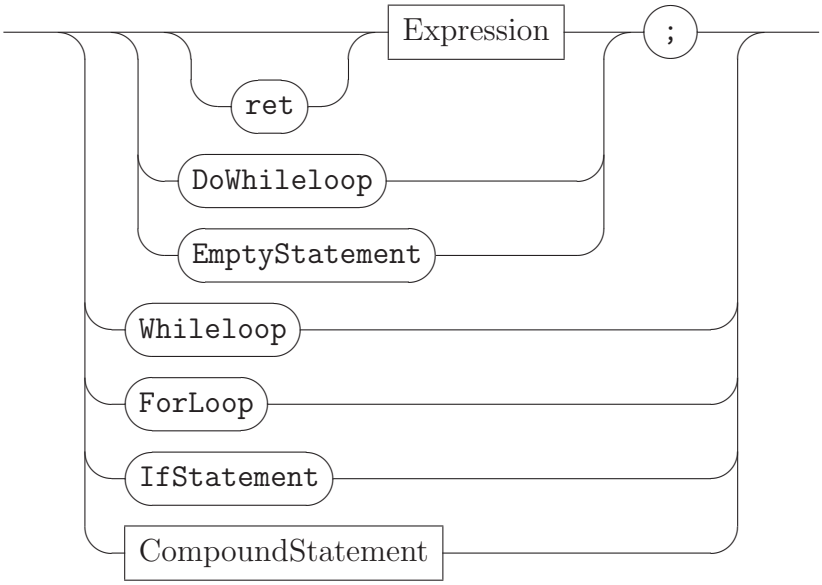
[2]

FunctionDeclaration

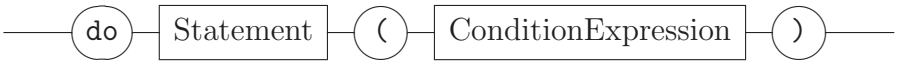


[2]Uwaga: zamiast <> na diagramie powinne stać znaki {}. Latexowi z niewiadomego mi powodu nie podoba się wykorzystanie nawiasów klamrowych w diagramach.

Statement

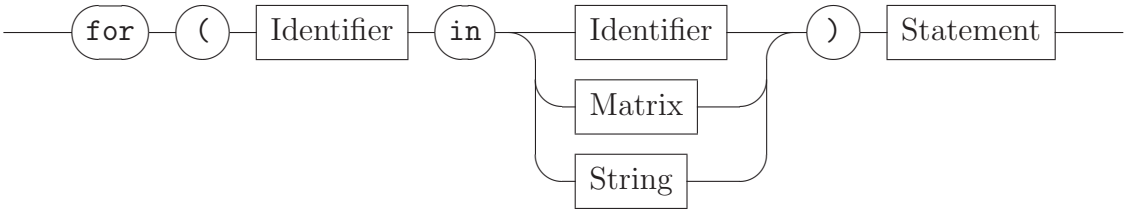


DoWhileLoop

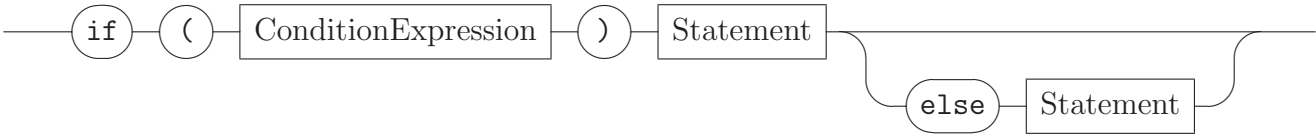


WhileLoop

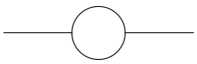
ForLoop



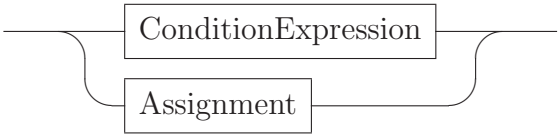
IfStatement



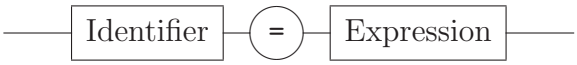
EmptyStatement



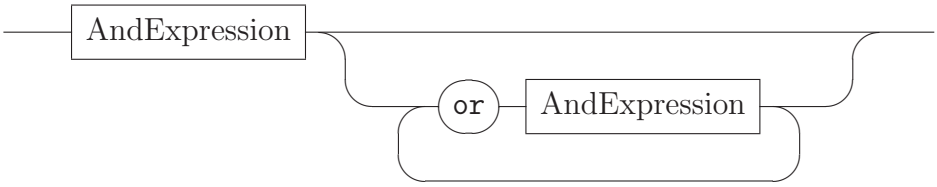
Expression



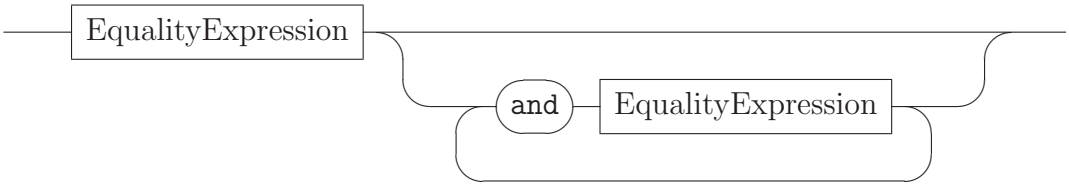
Assignment



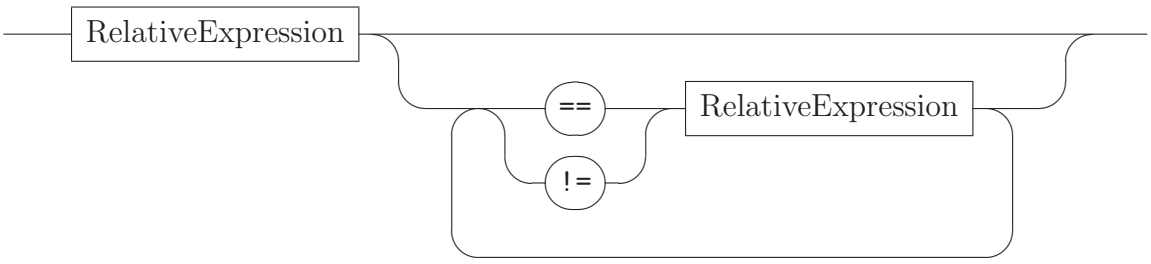
ConditionExpression



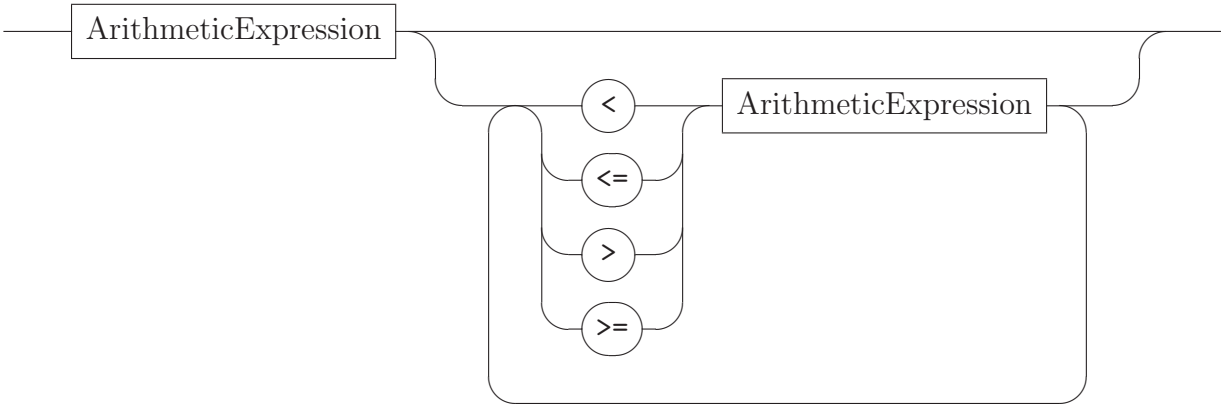
AndExpression



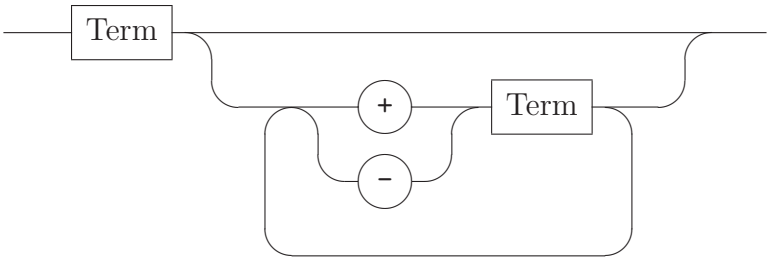
EqualityExpression



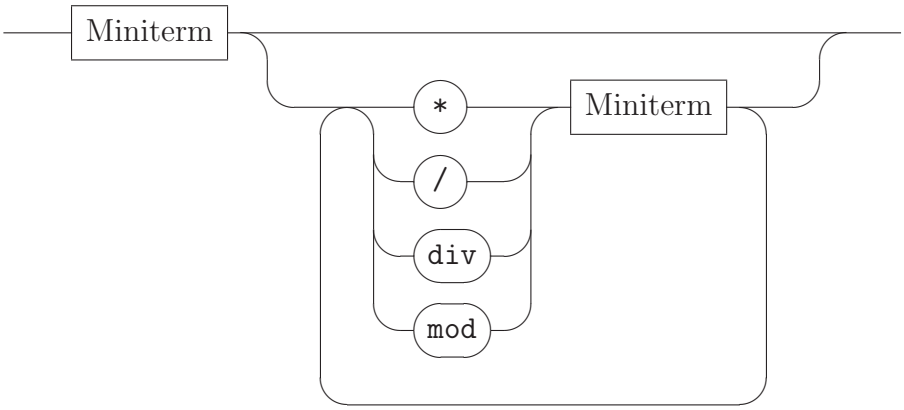
RelativeExpression



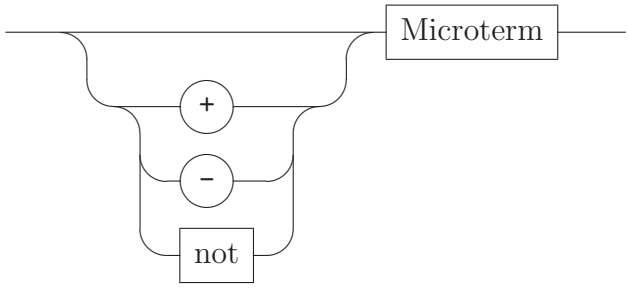
ArithmeticExpression



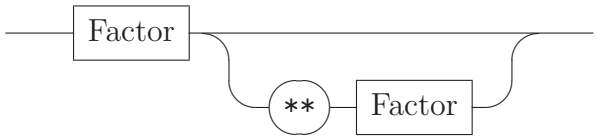
Term



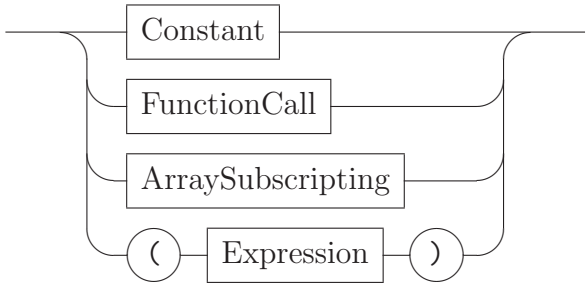
Miniterm



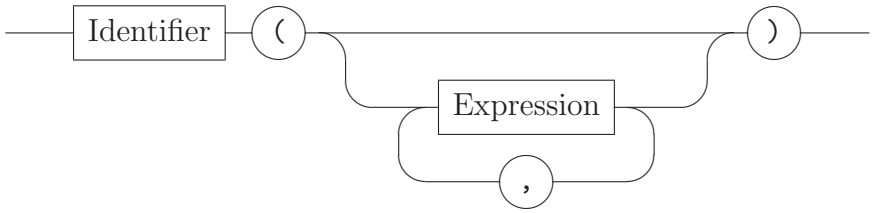
Microterm



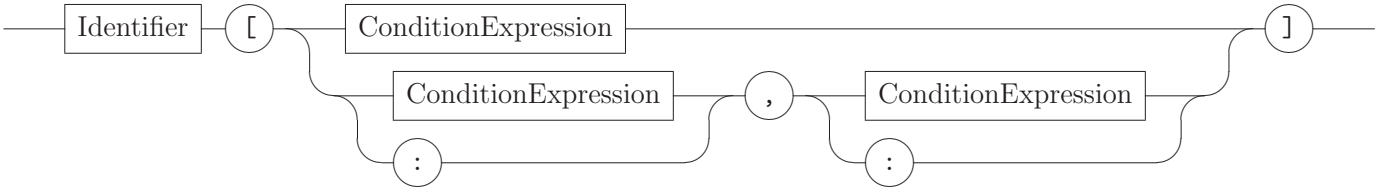
Factor



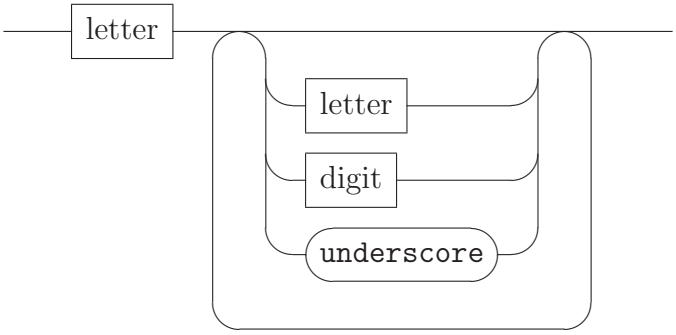
FunctionCall



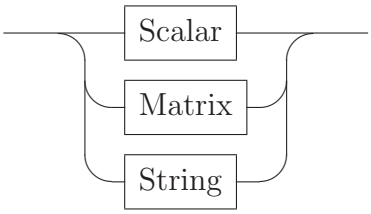
ArraySubscripting



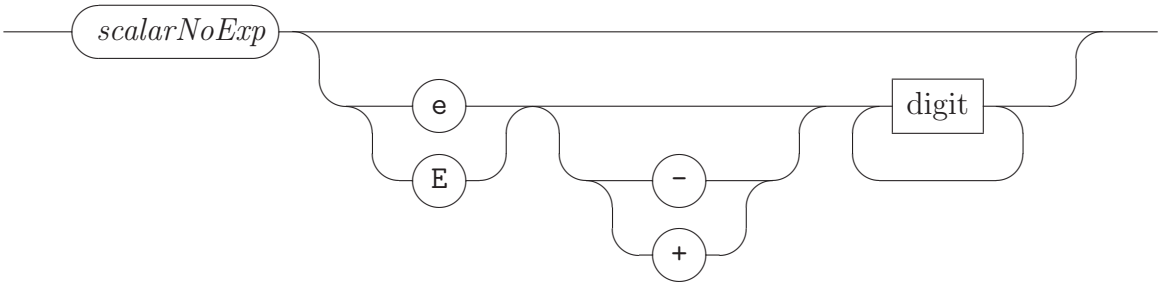
Identifier



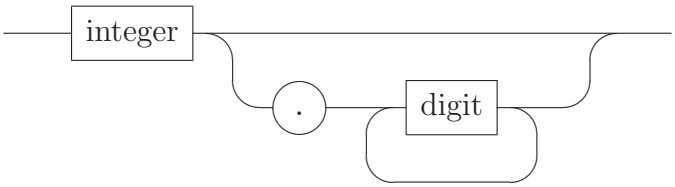
Constant



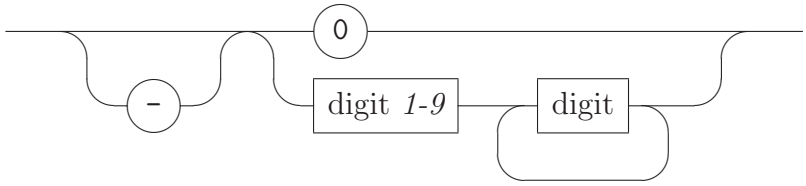
scalar



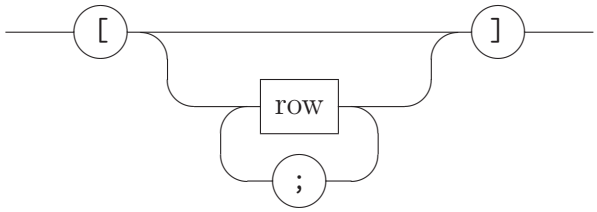
scalarNoExp



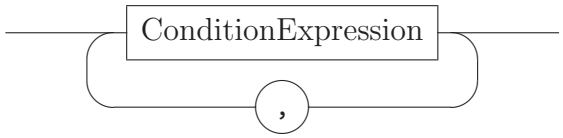
integer



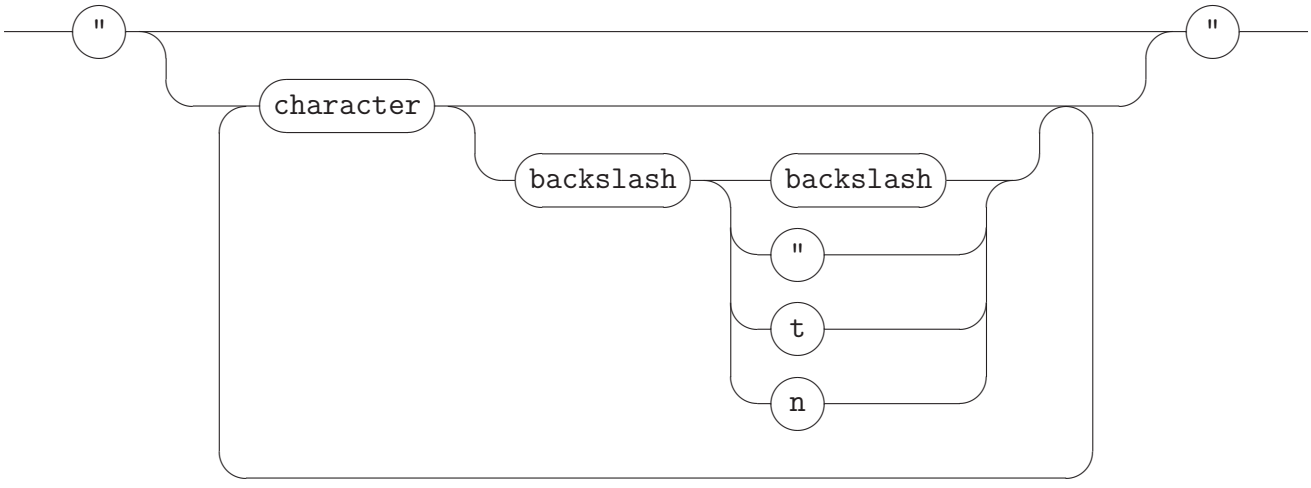
Matrix



row



string



gdzie *character* jest dowolnym znakiem oprócz ” i \ oraz znaków kontrolnych.

whitespace