TKOM - dokumentacja końcowa

Sebastian Pietras

1 Temat projektu

Stworzenie języka, w którym typem wbudowanym będzie czas oraz interpretera tego języka.

2 Opis języka

Żeby nadać językowi trochę charakteru, będzie on nosił nazwę: `Timon`

Język umożliwia przechowywanie informacji o:

- samej dacie (`date`)
- samej godzinie (`hour`)
- dacie i godzinie razem (`datetime`)
- różnicy czasu (`timedelta`)

Pozwala również na dokonywanie operacji na tych typach danych:

- `date1 - date2 = timedelta`
- `hour1 - hour2 = timedelta`
- `datetime1 - datetime2 = timedelta`
- `datetime - date = timedelta`
- `datetime - hour = timedelta`
- `date1 +- timedelta = date2`
- `hour1 +- timedelta = hour2`
- `datetime1 +- timedelta = datetime2`

- `timedelta1 +- timedelta2 = timedelta3`
- `timedelta1 */ number = timedelta2`

Operacje takie jak mnożenie dat nie miałyby sensu.

`date` przechowuje informacje o dniu, miesiącu i roku (np. `date = 10.04.2012`), `hour` o godzinie, minucie i sekundzie (np. `hour = 16:34:57`), `datetime` to połączenie `date` i `hour` (np. `datetime = 10.04.2012~16:34:57`), a `timedelta` o latach, miesiącach, tygodniach, dniach, godzinach, minutach i sekundach (np. `timedelta = '10Y 1W 5h'`).

Formaty literałów:

- `date` - `{dzień}.{miesiąc}.{rok}`
- `hour` - `{godzina}:{minuta}:{sekunda}`
- `datetime` - `{date}~{hour}`
- `timedelta` - `'{lata}Y {miesiące}M {tygodnie}W {dni}D {godziny}h {minuty}m {sekundy}s'`

Wszystkie jednostki czasu w `timedelta` są opcjonalne, `''` oznacza wartość zerową.

Bardziej formalny opis znajduje się w gramatyce opisanej przez EBNF w sekcji Gramatyka.

Daty są takie jak w zwykłym kalendarzu gregoriańskim. Uwzględnione są lata przestępne. Wszystkie lata muszą być z naszej ery.

Czas jest 24-godzinny.

Dostępna jest pętla `from`, która pozwala iterować między dwiema wartościami czasowymi podaną jednostką czasu. Na przykład:

```
from 01.01.2020 to 10.01.2020 by days as it
```

wykona się 10 razy i w każdym przebiegu zmienna `it` będzie przechowywać kolejny dzień.

Pozostałe cechy języka:

- dynamiczne typowanie
- typ liczbowy i znakowy
- instrukcja `print`
- instrukcja warunkowa `if else`
- definiowanie funkcji na poziomie globalnym
- deklarowanie zmiennych na każdym poziomie
- operacje logiczne i matematyczne o różnych priorytetach
- obsługa komentarzy wieloliniowych (między znakami `#`)
- dostęp do szczegółowych informacji o czasie (np. `date.days`)
- operacje logiczne mogą być wykonywane na wartościach matematycznych i wartości matematyczne mogą przyjmować wartości logiczne

3 Przykłady wykorzystania języka

Przykład 1

```
fun printDaysBetweenDates(d1,d2)
{
    # saving to d3 each consecutive day between d1 and d2 #
    from d1 to d2 by days as d3
    {
        print d3;
    };

    return 0;
};

var dt = 10.04.2018~10:57:00; # moment in time #
var d = 12.04.2018; # date #

printDaysBetweenDates(dt, d);
```

Przykład 2

```
var start_time = 10.06.2020;
var delay = '5D 0s';

var prev_t1 = 25.05.2020;
var prev_t2 = 20.05.2020;

if start_time + (prev_t1 - prev_t2) + delay <= 20.06.2020
{
    print "we have time";
}
else
{
    print "we dont have time";
};
```

Przykład 3

```
fun getMonthDiff(d1,d2)
{
    return (d1 - d2).months + 12 * (d1 - d2).years;
};

fun getFirstDateOfNextMonth(date)
{
    from date to date + '1M' by days as d
    {
        if d.days == 1
        {
            return d;
        };
    };

    return date;
};

print getMonthDiff(getFirstDateOfNextMonth(15.11.2021),
      getFirstDateOfNextMonth(28.05.2020));
```

Przykład 4

```
var t1 = 15:57:23;
var t2 = 20:45:00;
var td = t2 - t1;
var h = td.hours; # accessing specific timedelta info #
var napis = "hours between " + t1 + " and " + t2 + " : ";

print napis + h;
```

4 Gramatyka

```
program = { functionDefStatement |
            identifierFirstStatement |
            variableDefinitionStatement |
            ifStatement |
            fromStatement |
            printStatement |
            returnStatement  } ;

identifierFirstStatement = identifier, ( parameters | assignment ) ;
functionDefStatement = "fun", identifier, parametersDeclaration, body ;
variableDefinitionStatement = "var", identifier, [ assignment ] ;
ifStatement = "if", expr, body, [ "else", body ] ;
fromStatement = "from", fromRange, fromStep, fromIterator, body ;
printStatement = "print", expr ;
returnStatement = "return", expr ;

parametersDeclaration = "(", [ identifier, { ",", identifier } ], ")" ;
body = "{", { identifierFirstStatement |
              variableDefinitionStatement |
              ifStatement |
              fromStatement |
              printStatement |
              returnStatement }, "}" ;

assignment = assignmentOperator, expr ;

fromRange = expr, "to", expr ;
fromStep = "by", "years" |
                 "months" |
                 "weeks" |
                 "days" |
                 "hours" |
                 "minutes" |
                 "seconds" ;
fromIterator = "as", identifier ;
```

```
expr = logicAndExpr, { orOperator, logicAndExpr } ;
logicAndExpr = logicEqualExpr, { andOperator, logicEqualExpr } ;
logicEqualExpr = logicRelExpr, [ equalityOperator , logicRelExpr ] ;
logicRelExpr = logicTerm, [ relationalOperator, logicTerm ] ;
logicTerm = [ logicNegOperator ], mathExpr ;
mathExpr = multMathExpr, { additiveOperator, multMathExpr } ;
multMathExpr = mathTerm, { multiplicativeOperator, mathTerm } ;
mathTerm = [ mathNegOperator ],
           ( value | parenthesisedExpr ),
           [ timeInfoAccess ];

parenthesisedExpr = "(", expr, ")" ;

equalityOperator = equalOperator | notEqualOperator ;
relationalOperator = greaterOperator |
                     greaterOrEqualOperator |
                     lessOperator |
                     lessOrEqualOperator ;
additiveOperator = plusOperator | minusOperator ;
multiplicativeOperator = multiplyOperator | divisionOperator ;

assignmentOperator = "=" ;
orOperator = "|" ;
andOperator = "&" ;
equalOperator = "==" ;
notEqualOperator = "!=" ;
greaterOperator = ">" ;
greaterOrEqualOperator = ">=" ;
lessOperator = "<" ;
lessOrEqualOperator = "<=" ;
logicNegOperator = "!" ;

plusOperator = "+" ;
minusOperator = "-" ;
multiplyOperator = "*" ;
divisionOperator = "/" ;
mathNegOperator = "-" ;

value = numberLiteral |
        stringLiteral |
        dateLiteral |
        timeLiteral |
        datetimeLiteral |
        timedeltaLiteral |
        identifierFirstValue ;

identifierFirstValue = identifier, [ parametersCall ] ;
parametersCall = "(", [ expr, { ",", expr } ], ")" ;
```

```
timeInfoAccess = ".", ( "years" |
                        "months" |
                        "weeks" |
                        "days" |
                        "hours" |
                        "minutes" |
                        "seconds" ) ;

numberLiteral = "0" | ( nonZeroDigit, { digit } ) ;
stringLiteral = '"', { stringCharacter }, '"' ;
dateLiteral = two_digits, ".", two_digits, ".", four_digits ;
timeLiteral = two_digits, ":", two_digits, ":", two_digits ;
datetimeLiteral = dateLiteral, "~", timeLiteral ;
timedeltaLiteral = "'", [ numberLiteral, "Y" ],
                        [ numberLiteral, "M" ],
                        [ numberLiteral, "W" ],
                        [ numberLiteral, "D" ],
                        [ numberLiteral, "h" ],
                        [ numberLiteral, "m" ],
                        [ numberLiteral, "s" ], "'" ;

identifier = nonDigitCharacter, { character } ;

nonDigitCharacter = letter | specialCharacter ;
character = nonDigitCharacter | digit ;

stringCharacter = ( ? all visible characters ? - '"' ) | '\"' ;


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" |
         "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" ;

specialCharacter = "_" ;

two_digits = 2 * digit ;
four_digits = 4 * digit ;

digit = "0" | nonZeroDigit ;
nonZeroDigit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

comment = '#', stringCharacter - '#', '#' ;
```

W gramatyce nie są uwzględnione białe znaki i komentarze, bo mogą występować wszędzie i będą po prostu pomijane przy parsowaniu (nie licząc w środku stringLiteral i timedeltaLiteral)

W gramatyce są ujęte tylko liczności cyfr przy datach. Na poziomie analizy leksykalnej wykonywane jest również faktyczne sprawdzenie poprawności daty (na przykład czy miesiąc ma 30 czy 31 dni albo czy jest to rok przestępny).

Aby zawrzeć znak " w stringLiteral należy go poprzedzić znakiem \. Jeśli po znaku \ nie wystąpi znak " to \ zostanie potraktowany dosłownie.

5 Struktura interpretera

Program jest napisany w języku Python. Na wejście będzie podawana ścieżka do pliku zawierającego skrypt do interpretacji (bardziej formalny opis uruchomienia znajduje się w sekcji Uruchomienie).

Interpretacja składa się z kilku etapów, opisanych poniżej. Dodatkowo dostępne będą pomocnicze moduły obsługi plików, błędów i środowiska.

5.1 Analiza leksykalna

Moduł: `lexical_analysis.py`

Celem tego etapu jest zmiana znaków pliku na tokeny. Dostępna jest klasa `Lexer`, która poprzez metodę `get` zwraca i konsumuje następny token lub poprzez metodę `peek` zwraca następny token bez konsumpcji. Przy konstrukcji obiektu leksera należy podać obiekt umożliwiający odczytywanie pliku (np. obiekt klasy `FileReader` z modułu `source_readers.py`).

Przykład użycia:

```
with FileReader('path/to/script.tim') as fr:
    lex = Lexer(fr)
    token = lex.get()
```

Jeśli wystąpi błąd, zostanie rzucony wyjątek `LexicalError` (zdefiniowany w module `error_handling.py`).

5.2 Analiza składniowa

Moduł: `syntax_nodes.py`

Celem tego etapu jest sprawdzenie, czy tokeny są zgodne z gramatyką i stworzenie drzewa składniowego. Głównym węzłem drzewa jest `Program`, który tworzy cały drzewo. Należy podać mu obiekt leksera.

Analizator leksykalny będzie zbierał znaki w tokeny i zwracały je na żądanie analizatora składniowego, aby jak najszybciej przerwać analizę po wystąpieniu błędu syntaktycznego.

Przykład użycia:

```
with FileReader('path/to/script.tim') as fr:
    program = Program(Lexer(fr))
```

Jeśli wystąpi błąd, zostanie rzucony wyjątek `SyntacticError` (zdefiniowany w module `error_handling.py`).

5.3 Wykonanie

Moduł: `syntax_nodes.py`, `execution.py`

Celem tego etapu jest wykonanie po kolei instrukcji z drzewa zgodnie z ich znaczeniem. Każdy węzeł w drzewie definiuje swoje zachowanie podczas uruchomienia. Aby wykonać program, należy wywołać metodę `execute` głównego węzła, podając jako argument obiekt klasy `Environment` z modułu `execution.py`.

Przykład użycia:

```
with FileReader('path/to/script.tim') as fr:
    program = Program(Lexer(fr))

program.execute(Environment())
```

Jeśli wystąpi błąd, zostanie rzucony wyjątek `ExecutionError` (zdefiniowany w module `error_handling.py`).

6 Uruchomienie

Aby uruchomić interpreter należy posiadać Pythona w wersji co najmniej 3.6 i wywołać polecenie z katalogu głównego:

```
python3 -m timoninterpreter
        [-stage {lexer, parser, execution}]
        PATH_TO_SCRIPT
```

Przykładowe skrypt można znaleźć w `tests/acceptance/scripts/`.

Podając argument `-stage` wykonanie zostanie zatrzymane na podanym etapie

i zostaną podane informacje dotyczące stanu analizy na tym etapie. Domyślnym etapem jest `execution`.

Jeśli wystąpi błąd zostanie wypisana informacja o nim oraz o miejscu w kodzie, w którym wystąpił.

7 Testowanie

Dostępne jest prawie 500 testów jednostkowych dotyczących wszystkich etapów w katalogu `tests/unit`. Aby uruchomić je wszystkie należy wywołać z katalogu głównego:

```
python3 -m unittest discover tests/unit
```

Przygotowane zostały również testy akceptacyjne dotyczące 4 przykładowych skryptów i zachowania interpretera w każdym etapie. Znajdują się w katalogu `tests/acceptance`. Aby je uruchomić należy wywołać z katalogu głównego:

```
python3 -m unittest discover tests/acceptance
```