Język z jednostkami

dokumentacja końcowa

Joanna Sokołowska

2 czerwca 2021

Spis treści

1 Opis

Interpreter prostego języka umożliwiającego przetwarzanie zmiennych zawierających wartości liczbowe z dowolnymi jednostkami. Oferuje możliwość deklaratywnego definiowania jednostek i zależności między nimi. Jednostki mogą być poddawane operacjom arytmetycznym i są wykorzystywane przy określaniu poprawności operacji.

2 Założenia

- Język udostępnia typy wbudowane: int, float, string, bool, bazowe typy jednostek: kilo, meter, second oraz typ generyczny compound służący do obsługi jednostek złożonych, a także void jako typ wartości zwracanej z funkcji.
- Język pozwala na definiowanie własnych jednostek o typie prostym lub złożonym. Jednostki złożone mogą być definiowane jako iloczyn całkowitoliczbowych potęg jednostek prostych (wbudowanych lub zdefiniowanych przez użytkownika).
- Definicja jednostki złożonej wykorzystującej zdefiniowaną przez użytkownika jednostkę prostą musi znajdować się po definicji tej jednostki prostej.
- Język pozwala na definiowanie konwersji pomiędzy jednostkami, jednak nie dba aby konwersja z a $a \to b$ i $b \to a$ były symetryczne (tj. zastosowanie konwersji $a \to b \to a$ nie musi dawać takiej samej wartości jaka była na początku).
- Język umożliwia działania arytmetyczne na jednostkach, lecz przed ich wykonaniem sprawdza poprawność typów. Więcej o działaniach na jednostkach w sekcji *Operacje arytmetyczne*.
- Język pozwala na rzutowanie pomiędzy typami za pomocą zdefiniowanych konwersji, konwersji wbudowanych i w niektórych przypadkach konwersji niejawnych, jednak nie umożliwia rzutowania z każdego typu na każdy. Więcej o konwersjach w sekcji *Kilka słów o niejawnym rzutowaniu*.
- Na potrzeby obliczania wyrażeń logicznych wartości typu int i float są traktowane jako prawda jeśli są większe od 0 i jako fałsz w pozostałych przypadkach. Analogicznie - typy jednostkowe traktowane są jako prawda jeśli powiązana z nimi wartość numeryczna jest większa od 0.
- Interpreter ignoruje białe znaki.
- Kod programu musi znajdować się w całości w jednym pliku. Plik ten musi zawierać funkcję main zwracającą typ int i nie przyjmującą żadnych argumentów.
- Identyfikatory definiowane przez użytkownika nie mogą mieć takiej samej postaci jak słowa kluczowe języka.

3 Funkcjonalność

- Obsługa instrukcji warunkowej if... else....
- Obsługa pętli while, a także instrukcji break i continue.
- Możliwość dokonywania operacji arytmetycznych na jednostkach i typach numerycznych (dodawanie, odejmowanie, mnożenie, dzielenie i podnoszenie do potęgi), a także zmianę kolejności wykonywania działań poprzez zastosowanie nawiasów.
- Możliwość definiowania własnych typów jednostek w formie iloczynów dowolnych całkowitoliczbowych potęg jednostek bazowych (wbudowanych lub zdefiniowanych przez użytkownika).

- Możliwość definiowania zależności pomiędzy jednostkami w formie wyrażenia arytmetycznego wykorzystującego mnożenie, potęgowanie, dodawanie, dzielenie, odejmowanie, negację i nawiasowanie, a także stałe liczbowe i zmienne o typach jednostkowych.
- Przy dokonywaniu operacji arytmetycznych język sprawdza poprawność typów jednostek i wyświetla stosowne komunikaty o błędach.
- Obsługa instrukcji wbudowanych: `type` (podaje typ zmiennej) i `print`.
- Możliwość definiowania własnych funkcji.
- Obsługa wyrażeń logicznych z zastosowaniem alternatywy, koniunkcji i negacji oraz operacji porównania `==, !=, =<, <, >, >=`.
- Obsługa ograniczonego zakresu zmiennej.
- Możliwość przeciążania funkcji i konwersji.

4 Poradnik użytkownika

Składnia języka jest w większości podobna do tej znanej z języka C. Została okrojona ze względu na założony zakres funkcjonalności, a nowe elementy, godne uwagi są wyszczególnione w tym poradniku.

4.1 Uruchamianie

Interpreter dostarczany jest jako plik .jar. Do jego uruchomienia potrzebna jest Java SE w wersji 16 lub nowszej. Interpreter przyjmuje jeden argument - ścieżkę do pliku tekstowego zawierającego kod programu użytkownika. Uruchomienie programu:

```
java -jar interpreter.jar <path-to-code-file>
```

4.2 „Hello World!"

```
int main () {
    print("Hello world");
    return 0;
}
```

Kod prostego programu *Hello world* składa się z 3 części - definicji funkcji main, instrukcji wypisującej napis na standardowe wyjście i instrukcj powrotu, zwracającej 0;

4.3 Typy jednostkowe

Główną możliwością oferowaną przez język jest obsługa typów jednostkowy - tj. wartości numerycznych, z którymi powiązana jest jakaś jednostka. Język oferuje 3 wbudowane typy jednostkowe - `kilogram`, `meter` i `second`, automatyczne wykrywanie typów zmiennych jednostkowych zadeklarowanych jako `compound`, a także możliwość definiowania własnych jednostek, zgodnych z potrzebami użytkownika.

4.3.1 Typy definiowane przez użytkownika

Typy jednostkowe dzielą się na dwa rodzaje - proste i złożone. Typy proste charakteryzują się niepodzielnością - nie istnieją żadne zależnośći pomiędzy typem prostym, a innym typem jednostkowym, z wyjątkiem tych, które użytkownik zdefiniuje w formie konwersji. Wszystkie typy wbudowane są typami prostymi. Typy złożone natomiast to typy, które wyrażane są jako iloczyn typów prostych w dowolnych potęgach całkowitoliczbowych. Typy bazowe co do zamysłu odpowiadają jednostkom podstawowym z układu SI - nie da się ich zdekomponować na inne jednostki, a typy złożone - jednostkom pochodnym.

```
/* definiowanie jednostek prostych */
unit farenheit;
unit kiloMeter;

/* definiowanie jednostek zlozonych */
unit joule as  <kilogram * meter^2 / second ^2>;
unit newton as <kilogram * meter /  second ^2>;

int main(){
    ...
    /* definiowanie zmiennych jednostkowych */
    joule j = 12;
    newton n = 10;
    farenheit f = 13 + 20;
    kiloMeter km = 18;
    ...

}
```

Deklaracje typów muszą znaleźć się przed deklaracjami funkcji, a ponadto każdy typ jednostkowy musi być zdefiniowany przed wykorzystaniem, dlatego nieprawidłowa jest następująca sekwencja deklaracji:

```
unit amper;

unit om as <vat/amper>;     //vat jeszcze niezdefiniowany

unit vat;
...
```

4.3.2 Operacje arytmetyczne

Korzystanie z typów jednostkowych wprowadza dodatkowe ograniczenia związane z dozwolonymi operacjami arytmetycznymi.

Dodawanie i odejmowanie
Dodawanie i odejmowanie możliwe jest tylko dla dwóch jednostek o równoważnych typach. Nie ma możliwości dodania wartości liczbowej, niepowiązanej z żadną jednostką i nie są przeprowadzane żadne niejawne konwersje. Wynik tej operacji ma taki sam typ jak argumenty.

Mnożenie i dzielenie
Mnożenie i dzielenie możliwe jest dla dwóch dowolnych typów jednostkowych, a także dla typu jednostkowego i typu liczbowego. Wynikiem takiego działania jest zazwyczaj złożony typ jednostkowy wyliczany poprzez przemnożenie (lub podzielenie) typów argumentów. W przypadku gdy w wyniku tych wyliczeń wszystkie jednostki się skrócą, wynik będzie typem liczbowym.

```
unit farenheit;
unit joule as  <kilogram * meter^2 / second ^2>;
unit newton as <kilogram * meter /  second ^2>;

int main(){
    ...
    joule j = 2;
    newton n = 3;
```

```
    j * n;      // 6, typ = <kilogram ^2 * meter^3 * second^-4>
    j / n;      // 0.6666666, typ = <meter^1>
    j*2;        // 4, typ= joule <kilogram^1 * meter^2 / second ^2>

    meter m = 2;
    farenheit f = 1;

    m / m;        // 1, typ = int
    m / (3* m);   // 0.333333, typ = float
    m * f;        // 2, typ = <farenheit^1 * meter^1>
    ...
}
```

Potęgowanie

Typy jednostkowe mogą być podnoszone do potęgi, o ile wykładnikiem jest liczba całkowita. Wynik potęgowania to złożony typ jednostkowy z wykładnikami przy poszczególnych składnikach przemnożonych poprzez wykładnik.

```
unit newton as <kilogram * meter /  second ^2>;

int main(){
    ...
    newton n = 3;

    n ^ 2;      // 9, typ = <kilogram^2 * meter^2 * second ^-4>

    ...
}
```

4.3.3 Wyrażenia logiczne

Na potrzeby wyrażeń logicznych typy jednostkowe zachowują się jak ich numeryczne odpowiedniki tj. typy o wartości liczbowej większej od 0 odpowiadają wartości `true`, a równe lub mniejsze od 0 - wartośći `false`. Na potrzeby operacji relacyjnych zmienne jednostkowe są traktowane jak ich numeryczne odpowiedniki.

4.3.4 Konwersje

Język pozwala na definiowanie konwersji pomiędzy dowolnymi jednostkami. Składnia konwersji przypomina składnię funkcji o identyfikatorze będącym nazwą typu jednostkowego (wbudowanego, bazowego zadeklarowanego przez użytkownika lub zadeklarowanego typu złożonego). Konwersja może przyjmować dowolnie wiele argumentów będących dowolnymi typami jednostkowymi. Dla jednej jednostki można zdefiniować dowolnie wiele konwersji, jednak muszą się one różnić typami przyjmowanych argumentów.

W ciałem konwersji jest jednostkowe wyrażenie arytmetyczne, w którym możliwe są wszystkie obsługiwane operatory arytmetyczne (dodawanie, odejmowanie, mnożenie, dzielenie, potęgowanie), a którego składnikami są stałe liczbowe lub zmienne jednostkowe (argumenty funkcji). Konwersja zwraca wartość tego wyrażenia jako wartość typu jednostkowego podanego jako identyfikator konwersji. Na potrzeby wyliczania wartości wyrażenia jednostkowego wszystkie zmienne jednostkowe zachowują się jak ich odpowiedniki numeryczne (tj. jednostki nie są używane w określaniu poprawności i typu wynikowego).

Język oferuje również wbudowane konwersje, które pozwalają rzutować typy jednostkowe na typy numeryczne.

```
unit farenheit;
unit celsius;
unit kelvin;

/* definiowanie konwersji */
let kelvin as (celsius c) { c + 273.15};
let farenheit as (celsius c) { c * 9/5 + 32};
let kelvin as (farenheit f) { (f-32) * 5/9 + 273.15 };

int main(){
    ...
    celsius c = 10;

    /* korzystanie ze zdefiniowanych konwersji */
    farenheit f = farenheit(c);     // 50, typ = farenheit
    farenheit f2 = farenheit(2 *c); // 68, typ = farenheit
    kelvin k = kelvin(c);           // 283.15, typ = kelvin
    kelvin k2 = kelvin(f);          // 283.15, typ = kelvin

    /* korzystanie z konwersji domyslnych */
    int i = int(k2);                // 283, typ = int
    float f = int(k);               // 283.15, typ = float
    ...
}
```

4.3.5 Zmienne o typie wykrywanym

W celu ułatwienia zapisywania wyników operacji na typach jednostkowych i ulżenia użytkownikowi w konieczności monotonnego definiowania typów jednostkowych potrzebnych do opisania takich zmiennych, język oferuje typ `compound`. Typ zmiennej zadeklarowanej jako `compound` zostanie automatycznie wykryty w trakcie wykonania programu, gdy zmiennej zostanie przypisana wartość. W związku z tym nie może być zadeklarowany jako parametr funkcji, konwersji ani jako wartość zwracana przez funkcję. Nie może zostać mu też przypisana sama wartość liczbowa (bez typu jednostkowego) i próby takich przypisań będą skutkowały zgłoszeniem wyjątku przez program. Jeśli użytkownik chciałby potem sprawdzić, jaki typ został przypisany zmiennej, może wykorzystać w tym celu funkcję wbudowaną `type`.

```
unit joule as  <kilogram * meter^2 / second ^2>;
unit newton as <kilogram * meter /  second ^2>;

int main(){
    joule j = 12;
    newton n = 6;

    compound c1 = j * n;
    compound c2 = j / n;

    type(c1);               //compound <meter^3*kilogram^2*second^-4>
    type(c2);               //compound <meter^1>

    return 0;
}
```

4.4 Kilka słów o niejawnym rzutowaniu

Przypadki, w których zostanie dokonana próba niejawnego rzutowania:

- Przypisanie wartości do zmiennej o typie jednostkowym. Powiedzie się jeśli wyrażenie, które użytkownik chce przypisać do zmiennej ma typ int lub float.
- Przekazywanie parametrów do funkcji `print` - wartość dowolnego typu zostanie zmieniona na napis.
- Zwracanie parametru funkcji - zostanie podjęta próba rzutowania na typ zwracany. Powiodą się rzutowania typów numerycznych na jednostkowe i odwrotnie, dowolnego typu na napis, typu numerycznego lub jednostkowego na wartość logiczną. W pozostałych przypadkach zostanie zgłoszony wyjątek.

4.5 Niestandardowe operatory

Na wzór języka python operator * zastosowany do typów int i string powoduje n-krotne powielenie napisu.

5 Struktura projektu

5.1 Moduły

1. **Źródło**. Podaje kolejne znaki kodu i znak specjalny oznaczający koniec źródła.
2. **Analizator Leksykalny**. Pobiera znaki ze źródła i rozpoznaje tokeny zgodnie ze zdefiniowaną gramatyką.
3. **Analizator Składniowy**. Parser rekursywny zstępujący odpowiadający za budowanie drzewa programu na podstawie gramatyki języka. Ponadto weryfikuje też czy typy zostały zdefiniowane przed użyciem.
4. **Interpreter**. Moduł wykonujący program. Odpowiada również za analizę semantyczną. Składa się z dwóch głównych klas: `Interpreter`, wykonujący program i klasa `Casting` odpowiadająca za dokonywanie konwersji typów, wyliczanie typów działań arytmetycznych i dokonywanie większości obliczeń matematycznych.
5. **Moduł pomocniczy**. Zawiera klasy użytkowe i klasy wyjątków.

5.2 Struktury danych

1. **Drzewo programu**. Struktura produkowana przez analizator składniowy. Reprezentuje drzewo rozbioru programu za pomocą drzewa obiektów. Przkazywana jest do interpretera i na jej podstawie wykonywane są instrukcje.
2. **Position**. Struktura przechowywująca pozycję tokenu w tekście źródła.
3. **Token**. Klasa reprezentująca tokeny wraz z ich typem, wartością i pozycją.

5.3 Obsługa błędów

Wszystkie zauważone nieprawidłowości wyświetlane są użytkownikowi w postaci wyjątków. Analizator leksykalny wskazuje linię i kolumnę, w której napotkał znak błędny wraz ze stosownym komunikatem dostarczającym dalsze informacje o błędzie np. „Missing |" lub „Improper floating point number". Analizator składniowy wskazuje na początek niespodziewanego tokenu, podaje rozpoznany typ tokenu, a także typ tokenu, którego spodziewał się w tym miejscu. Interpreter zwraca dwa rodzaje wyjątków. Są to: wyjątki związane z nieprawidłowym rzutowaniem typów, w których wyświetlana jest

informacja o typie wartości rzutowanej i typie, na który próbowano ją zrzutować; oraz inne wyjątki związane z niepoprawnym semantycznie programem. W obu przypadkach podawany jest numer linijki, który spowodował wyjątek, a jeśli jest to możliwe to także dodatkowe informacje o błędzie.

5.4 Testowanie

Dla wszystkich modułów zostały napisane testy jednostkowe, a dla wybranych fragmentów interpretera również testy integracyjne. Całkowite pokrycie testami wynosi .... . Po pobraniu źródeł projektu testy można uruchomić poleceniem `mvn test`, a także wygenerować raport dotyczący pokrycia kodu testami wykorzystując polecenie `mvn clean verify`. Raport można obejrzeć otwierając plik ./target/site/index.html.

6 Gramatyka

```
module grammar;
# main part
program : {unit_dcl | conversion_dcl} function_def {function_def};
function_def : type identifier parameters block_stmt;
parameters : "(" [type identifier {"," type identifier}] ")" ;
block_stmt : "{" [stmt  {stmt}] "}";
stmt: block_stmt | return_stmt | loop_stmt | call_stmt | if_stmt | print_stmt | explain_stmt  |
| var_dcl_stmt | break_stmt | continue_stmt | type_check_stmt;

# flow control statements
return_stmt : "return" ( identifier | expr) ";";
loop_stmt : "while" "(" expr ")" stmt ;
call_stmt: identifier "(" args ")" ";";
args: [expr {"," expr}];                #arguments
if_stmt : "if" "(" expr ")" stmt ["else" stmt];
break_stmt : "break" ";";
continue_stmt : "continue" ";";

# bulit-in statements
print_stmt : "print" "(" args ")" ";";
type_check_stmt: "type" "(" identifier ")";
assign_stmt : identifier "=" expr ";";
var_dcl_stmt : type (identifier ";") | [assign_stmt] ;

# defining custom units
unit_dcl : "unit" identifier ["as" compound_expr ] ";";
# conversionFunction expression for compound unit definition
# only one division is possible
compound_expr: "<" compound_term ["/" compound_term] ">" ";";
compound_term: compound_exp {"*" compound_exp};
compound_exp: unit_type ["^" exponent];
exponent: ["-"] non_zero_number;

# defining unit relationships
conversion_dcl: "let" unit_type unit_parameters conv_fun;
# expressions limited to arithmetic operations
conv_fun : "{" conv_unit_expr "}" ";";
conv_unit_expr: multi_unit_expr {add_op multi_unit_expr};
multi_unit_expr: power_unit_expr {multi_op power_unit_expr};
```

```
power_unit_expr: {unary_unit_expr "^"} unary_unit_expr;
unary_unit_expr: ["-"] unit_expr;
unit_expr: "(" conv_unit_expr ")" | unit_value;
unit_value: number | identifier;
unit_parameters: "(" [unit_type identifier {"," unit_type identifier}] ")" ;


# operator precedence
expr : unit_type "(" ( or_expr  | or_expr ) ")";
or_expr : and_expr { "||" and_expr};
and_expr : comp_expr {"&&" comp_expr};
comp_expr : rel_expr {comp_op rel_expr};
rel_expr : arithmetic_expr {rel_op arithmetic_expr};
arithmetic_expr: multi_expr {add_op multi_expr};
multi_expr: power_expr {multi_op power_expr};
power_expr: {unary_expr "^"} unary_expr;
unary_expr: {unary_op} not_unary_expr;
not_unary_expr: "(" expr ")" | value;

# operators
comp_op: "==" | "!=";
rel_op: "<="|">="|"<"|">";
add_op: "+" | "-";
multi_op: "/" | "*" ;
unary_op: "!"|"-";

# values
value : call_stmt |  literal | identifier;
identifier : ((underscore  (letter | digit)) | letter) {letter | digit | underscore};
literal: base_type| string | number;
#types
type: base_type | unit_type;
base_type: "int" | "float" | "bool" | "string" | "void";
unit_type: base_unit | identifier | "compound";
base_unit: "kilogram" | "meter" | "second";
string: "\"" {character} "\"";
character: letter | digit |special_char;
letter: "a"-"z" | "A"-"Z";
special_char: underscore | "\\" escape_char | "." | "," | "(" | ")" | "{" | "}" | " " | ":" |"-"
escape_char: "t" | "n" | "\\";

number : integer ["." {"0"} non_zero_number];
integer : "0" | non_zero_number;
non_zero_number :  non_zero_digit {digit};
digit: "0" | non_zero_digit;
non_zero_digit: "1" - "9";
underscore: "_";
```

7 Przykładowe programy

Implementacja silni

```
int factorial(int n){
    if(n==1){
        return 1;
    }
    return n * factorial(n-1);
}
int main(){
    int i = 1;
    while(i <=10){
        print( i + "! = " + factorial(i));
        i = i + 1;
    }
    return 0;
}
```

Rozwiązanie zadania *Fizzbuzz*

```
unit joule as <kilogram * meter^2 / second ^2>;

int mod (int n, int by){
    if(by <= 0){
        return 0;
    }
    int res = n/by;
    return n - res * by;
}

int main(){
    joule  j = 1;
    while( j < 100){
        int mod3 = mod(int(j), 3);
        int mod5 = mod(int(j), 5);
        if(mod5 == 0){
            if(mod3 == 0){
                print("fizzbuzz");
            }else{
                print("buzz");
            }
        }else if(mod3 == 0){
            print("fizz");
        }else{
            print(int(j));
        }
        j = j + joule(1);
    }
    return 0;
}
```

Przykład wykorzystania jednostek do opisu wybranego fragmentu układu SI

```
unit amper;

unit volt as <meter^2 * kilogram / second^3 * amper>;
unit ohm as <volt/amper>;

ohm parallel_resistors ( ohm r1, ohm r2){
    return r1 * r2 / (r1 + r2);
}

ohm series_resistors (ohm r1, ohm r2){
    return r1 + r2;
}

int main (){
    ohm resistor1 = 20;
    ohm resistor2 = 40;

    amper a = 10;

    volt voltage_for_parallel = parallel_resistors(resistor1, resistor2) * a;

    compound voltage_for_series = series_resistors(resistor1, resistor2) * a;

    print(voltage_for_parallel);
    print(voltage_for_series);

    return 0;
}
```

Przykład wykorzystania złożonych jednostek i wyliczania typów

```
unit joule as <kilogram * meter^2 / second ^2>;
unit newton as <kilogram * meter /  second ^2>;

int main(){
    joule j = 12;
    newton n = 6;
    compound c1 = j * n;
    compound c2 = j / n;

    if(c1 > c2){
       print("if: ", c1, c2);
    }else if(2 > n){
        print("else if: ", c1, c2);
        while(c1 < c2){
            c1 = c1 * 2;
        }
    }else{
        print("else: ", c1, c2);
    }

    return 0;
}
```

Przykład wykorzystania kownwersji w celu realizacji różnych granularności danej zmiennej

```
unit hour;
unit minute;

let hour as (second s) { s/(60*60)};
let hour as (minute m) { m/60};

let minute as (second s) {s /60};
let minute as (hour h){ h*60 };

let second as (hour h) {h * 60 * 60};
let second as (minute m) {m * 60};

int main (){
    second s1 = 45;
    minute m1 = 150;
    hour h1 = 0.5;

    print(s1, minute(s1), hour(s1));
    print(second(m1), m1, hour(m1));
    print(second(h1), minute(h1), h1);

    return 0;
}
```