Język z jednostkami
dokumentacja wstępna

Joanna Sokołowska, nr indeksu 289463

11.04.2021

1 Treść zadania

Język umożliwiający podstawowe przetwarzanie zmiennych zawierających wartości liczbowe z dowolnymi jednostkami (np. SI). Może istnieć możliwość deklaratywnego zdefiniowania znanych jednostek i ich ewentualnych relacji lub te relacje mogą być predefinionwane (ale bardziej złożone). Jednostki powinny być 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.
- Język pozwala na definiowanie własnych jednostek o typie prostym lub złożonym. Jednostki złożone mogą być definiowane jako iloraz iloczynów całkowitoliczbowych potęg jednostek prostych.
- 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. Dodawanie i odejmowanie możliwe jest tylko dla jednostek tego samego typu, a mnożenie i dzielenie możliwe jest dla wszystkich typów jednostek (typ wyniku będzie typem generycznym).
- Język pozwala na rzutowanie, jednak wymaga dokonania go jawnie i nie obsługuje rzutowania z każdego typu na każdy. Obsługiwane są m.in. rzutowania pomiędzy typami numerycznymi (int, float) a bazowymi jednostkami i typem compound oraz tymi zdefiniowanymi przez użytkownika.
- Język pozwala na potęgowanie tylko przy całkowitoliczbowym wykładniku potęgi.
- 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.

- Język nie obsługuje komentarzy.
- Język nie obsługuje typów tablicowych, wskaźników ani operacji na pamięci.
- Identyfikatory definiowane przez użytkownika nie mogą mieć takiej samej postaci jak słowa kluczowe języka.

3 Funkcjonalności

- 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 całkowitoliczbowej), a także zmianę kolejności wykonywania działań poprzez zastosowanie nawiasów.
- Możliwość definiowania własnych typów jednostek w formie ilorazu 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: *explain* (podaje konwersję zdefiniowanego typu złożonego na typ generyczny) i *type* (podaje typ zmiennej).
- Możliwość definiowania własnych funkcji.
- Obsługa wyrażeń logicznych z zastosowaniem alternatywy, koniunkcji i negacji.
- Obsługa ograniczonego zakresu zmiennej.

4 Gramatyka

```
module grammar;
# main part
program : {unit_dcl | conversion_dcl} function_def {function_def};
function_def : type identifier arg_list block_stmt;
arg_list : "(" [type identifier {"," type identifier}] ")" ;
block_stmt : "{" [stmt {stmt}] "}";
stmt: block_stmt | return_stmt | loop_stmt | call_stmt | if_stmt | print_stmt |
    ↪ explain_stmt | assign_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}];
if_stmt : "if" "(" expr ")" stmt ["else" stmt];
break_stmt : "break" ";";
continue_stmt : "continue" ";";

# bulit-in statements
print_stmt : "print" "(" args ")" ";";
explain_stmt : "explain" "(" unit_type ")";
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 ] ";";
# conversion 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 ["^" non_zero_number];
unit_type: base_unit | identifier;

# defining unit relationships
conversion_dcl: "let" unit_type arg_list unit_expr";";
# expressions limited to arithmetic operations
unit_expr: multi_unit_expr {additive_op multi_unit_expr};
multi_expr: power_unit_expr {multi_op power_unit_expr};
power_unit_expr: unary_unit_expr {power_op unary_unit_expr};
unary_expr: ["-"] not_unary_expr;
not_unary_expr: "(" unit_expr ")" | unit_value;
unit_value: number | base_unit | 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 {additive_op multi_expr};
multi_expr: power_expr {multi_op power_expr};
power_unit_expr: unary_expr {"^" unary_expr};
unary_expr: {unary_op} not_unary_expr;
not_unary_expr: "(" expr ")" | value;

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

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

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