TKOM - Projekt Wstępny

Jeremi Sobierski 310901

Opis zakładanych funkcjonalności

Typy

W języku korzystać można zarówno z typów wbudowanych jak i zdefiniowanych przez użytkownika. Definicje typów mogą być generyczne. Każdy nowo zdefiniowany typ posiada prototyp, który definiuje dane w nim zawarte. Prototyp może być dowolnym innym typem).

Proste typy wbudowane:

1. `int`
2. `bool`
3. `char`
4. `float`
5. `String`

Złożone typy wbudowane:

1. krotka - `(...)`
2. wektor - `[...]`
3. strukutra - `{...}`

```
type Angle float

type Car {
    model: String,
    doors: int,
    fuel: float,
}
```

definicja własnego typu

Cechy

Użytkownik może definiować własne cechy oraz je implementować. Może również korzystać z implementacji do przeciążania pewnych operatorów.

```
trait Vehicle {
    fun drive()
    fun getFuel(): float
}

def Vehicle for Car {
    fun drive() { ... }
    fun getFuel(): float -> ...
}
```

definicja cechy oraz jej implementacja

Funkcje

Język pozwala na definicje nowych, potencjalnie generycznych funkcji. Argumenty do funkcji przekazywane są przez kopię.

```
fun testVehicle<V is Vehicle>(vehicle: V) {
    vehicle.drive();
    print("fuel after driving: " + vehicle.getFuel() as String);
}
```

Wywołanie funkcji lub metody odbywa się z użyciem nawiasów. Jeżeli metoda o podanej nazwie występuje w jendej implementowanej cesze, to należy jawnie wskazać jedną z nich.

```
register(car);
car.setFuel(42.0);
car.crashInto:<Truck>(truck);

// assuming dive is also in another implementation
car.<Vehicle>drive(truck);
```

Wyrażenia

Poza oczekiwanymi wyrażeniami (literały, wywołania itd.) dostępne są także następujące konstrukcje:

`if` - na podstawie warunku logicznego lub dopasowania ewaluuje wartość jednego z dwóch wyrażeń. Jeśli oba mają ten sam typ, przyjmuje wartość ewaluowanego wyrażenia.

`while` - wielokrotnie ewaluuje wyrażenie na podstawie warunku logicznego lub dopasowania. Jeśli warunkiem jest dopasowanie, w każdej kolejnej iteracji dopasowywana jest wartość ciała pętli z iteracji poprzedniej, a wartością wyrażenia `while` jest wartość ostatniej iteracji.

`match` - dopasowuje wartość wyrażenia do jednego z wzorców, a następnie ewaluuje i przyjmuje wartość przypożądkowanego dopasowanemu wzorcowi wyrażenia.

```
let result = if (condition) {
    print("was true");
    0
} else 1;

while (index < 10) {
    print("loop");
    index = index + 1;
};

let nums = [0, 0, 0, 1, 2, 3];
let trimmed = while (let [0, ..rest] = vector) rest;
// trimmed == [1, 2, 3]

let message = match car {
    { fuel: 0, model } => model + " has no more fuel!";
    { 0 < fuel < 0.2 } => "low on fuel";
    { fuel }           => fuel as String + " fuel left";
};
```

Zmienne

Język pozwala definiować zmienne mutowalne za pomocą słowa kluczowego `var` oraz niemutowalne za pomocą słowa kluczowego `let`. Typ zmiennej może być pominięty, zostanie on wtedy domniemany.

```
let num    = 3;
var mutNum = 4;

num    = 2; // error: let bindings cannot be reassigned
mutNum = 2; // OK
```

Gramatyka

Poniżej przedstawiona jest gramatyka języka

```
program             = functionDefinition
                    | typeDefinition
                    | traitDefinition
                    | traitImplementation;

functionDefinition  = signature, ( "->", expression | block );

typeDefinition      = "type", identifier, [ typeParameters ], typeName;
traitDefinition     = "trait", identifier, [ typeParameters ], traitBody;
traitImplementation = "def", [ typeParameters ], typeName, "as", traitName
                    , implementations;

nameAndType         = identifier, ":", typeName;

typeParameters      = "<", typeParameter, { ",", typeParameter }, [","], ">";
typeArguments       = "<", typeName, { ",", typeName }, [","], ">";
typeParameter       = identifier, [ "is", traitName, { "+", traitName } ];

typeName            = identifier, [ typeArguments ]
                    | "(", [  typeName, { ",",  typeName }, [","] ], ")"
                    | "{", [ nameAndType, { ",", nameAndType }, [","] ], "}"
                    | "[", typeName ,"]";

traitName           = identifier, [ typeArguments ];
implementations     = "{", { functionDefinition }, "}"
                    | functionDefinition;


signature           = "fun", identifier, [ typeParameters ], "("
                    , [ nameAndType, { ",", nameAndType }, [","] ]
                    , ")", [ ":", typeName ];

traitBody           = "{", { signature }, "}"
                    | signature;

tupleExpression     = expression, { ",", expression };
expression          = { controlFlow }, assignment;
assignment          = orExpression, [ "=", assignment ];
orExpression        = andExpression, { "||", andExpression };
andExpression       = equalityExpression , { "&&", equalityExpression };

equalityExpression  = relationalExpression
                    , { equalityOperator, relationalExpression };
```

```
relationalExpression     = additiveExpression
                         , { relationalOperator, additiveExpression };

additiveExpression       = multiplicativeExpression
                         , { additiveOperator, multiplicativeExpression };

multiplicativeExpression = castExpression
                         , { multiplicativeOperator, castExpression };

castExpression           = prefixExpression, { "as", prefixExpression };
prefixExpression         = { prefixOperator }, postfixExpression;
postfixExpression        = primaryExpression, { postfixOperator };
primaryExpression        = stringLiteral
                         | numericLiteral
                         | structLiteral
                         | identifier, [ structLiteral ]
                         | "[", ":", typeName", "]"
                         | "[", expression, { ",", expression }, [","] "]"
                         | "(", expression, ")"
                         | "(", [ expression, ",", [ tupleExpression, [","] ] ],")"
                         | "match", scrutinee, "{" matchCase, { matchCase } "}"
                         | block;

structLiteral            =  "{", [ structProperty, { ",", structProperty },
[","], ], "}"
structProperty           = identifier, [ ":", expression ];

block                    = "{", blockItem, ";", [ blockItem, { ";", blockItem } ],
[ ";" ] "}"

scrutinee                = identifier
                         | "(", expression, ")"
                         | "(", expression, ",", [ tupleExpression, [","] ], ")";

matchCase                = tuplePattern, "=>", matchArm, ";";

matchArm                 = tupleExpression
                         | exitFlowOperator, [ tupleExpression ];

prefixOperator           = "!" | "-";
postfixOperator          = ".", identifier, [ call ]
                         | ".", numericLiteral,
                         | ".", "match", "{", matchCase, { matchCase }, "}"
                         | ".", "<", traitName, ">", identifier, call,
                         | "[", tupleExpression, "]"
                         | call;

call                     = [ callTypeArguments ]
                         , "(" [ expression, { ",", expression }, [","] ] ")";

callTypeArguments        = ":<", typeName, { ",", typeName }, [","], ">";

blockItem   = "let" , destructureTuple, [ ":", typeName ], "=", tupleExpression
            | "var" , destructureTuple, [ ":", typeName ], "=", tupleExpression
            | exitFlowOperator, [ tupleExpression ]
            | expression;
```

```
equalityOperator        = "==" | "!=";
relationalOperator      = "<" | ">" | "<=" | ">=";
multiplicativeOperator  = "/" | "*" | "%";
additiveOperator        = "+" | "-";
exitFlowOperator        = "return" | "break" | "continue";

controlFlow             = "if", "(", condition, ")", expression, "else"
                        | "while", "(", condition, ")";

condition               = "let", tuplePattern, [":",typeName ],"=",tupleExpression;
                        | expression;

tuplePattern            = pattern, { ",", pattern };
pattern                 = "(" tuplePattern, [","] ")"
                        | "{", propertyBinding, { ",", propertyBinding }, [","] "}"
                        | "[", [ elementBinding, { ",", elementBinding },[","]],"]"
                        | stringLiteral
                        | identifier, [ halfBoundedNumeric ]
                        | numericLiteral, [ boundedNumeric ];



propertyBinding         = identifier, [ ":", pattern ]
                        | identifier, [ halfBoundedNumeric ]
                        | numericLiteral, boundedNumeric;

elementBinding          = pattern
                        | "..", [ identifier ];

halfBoundedNumeric      = relationalOperator, numericLiteral
                        | boundedNumeric;

boundedNumeric          = ("<" | "<="), identifier, [("<" | "<="), numericBound]
                        | (">" | ">="), identifier, [(">" | ">="), numericBound];

numericBound            = numericLiteral
                        | identifier;

destructureTuple        = destructure, { ",", destructure };

destructure     = "(" destructureTuple, [","] ")"
                | "{", destructureProperty, { ",", destructureProperty }, [","] "}"
                | "[", destructureElement, { ",", destructureElement }, [","] "]"
                | identifier;

destructureProperty     = identifier, [ ":", destructure ];
destructureElement      = destructure
                        | "..", [ identifier ];
```

Tokeny:

```
identifier      matches "[a-zA-Z][a-zA-Z0-9]*";
numericLiteral  matches "\d+(\.\d+)?";
stringLiteral   matches "\".*?[^\\]\"";
keywords        = "fun" | "type" | "def" | "trait" | "var" | "let" | "int"
                | "float" | "as" | "is" | "for" | "if" | "while" | "else"
                | "return" | "break" | "match";
```

```
punctuation     = ".." | "." | ":" | ":<" | "," | "<" | ">" | "<=" | ">=" | "=="
                | "!=" | "/" | "*" | "%" | "+" | "-" | "(" | ")" | "{" | "}" | "["
                | "]" | "=" | "!" | "&&" | "||";
```

Komunikaty Błędów

Przykłady komunikatów błędów napotkanych w trakcie kompilacji lub wykonania programu:

```
type error: expected value of type "String", has type "int" instead
  |
42| let text: String = 456;
  |                    ^^^
  @ main.ice:42:21

syntax error: expected ";", got "fun" instead
  |
42| let text: String = 456 fun
  |                        ^^^
  @ main.ice:42:25

index out of bounds error: accessing index 5 on a vector of size 3
  |
36| let element = vector[5];
  |                     ^^^
  @ main.ice:36:22

arithmetic error: type "int" has overflown
  |
56| index = index + 1;
  |         ^^^^^^^^^
  @ main.ice:56:9

arithmetic error: division by zero
  |
70| fraction = 3 / 0;
  |              ^^^
  @ main.ice:70:14
```

Sposób uruchomienia

Poniżej przykład komendy skompilowania oraz uruchomienia programu składającego się z dwóch plików źródłowych:

```
icy lib.ice main.ice -o ouputFile
./outputFile
```

Zwięzły opis realizacji

Główne komponenty

- lekser
- parser
- analiator sematyczny
- generator formy pośredniej LLVM

Lekser przekazuje strumień tokenów do parsera, który buduje na jego podstawie AST. Przechodzi przez nie następnie analizator sematyczny, sprawdzając poprawność typów, ograniczeń typów generycznych oraz odnajduje symbole. Po tym przez drzewo przechodzi generator formy pośredniej LLVM, która następnie kompilowana jest do pliku objektowego.

Zwięzły opis sposobu testowania

Przewidywane są zarówno testy jednostkowe do każdego z komponentów oddzielnie (lekser, parser, analizator sematyczny) oraz wspólne testy integracyjne.