AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
Wydział Fizyki i Informatyki Stosowanej

Praca inżynierska

Karol Etrych

kierunek studiów: **informatyka stosowana**

Opiekun: dr inż. Janusz Malinowski

Kraków, styczeń 2017

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy.

.....

(czytelny podpis)

1. Spis treści

2. Wstęp

1.1. Założenia projektu.

Planowane cechy języka to:

- Statyczne typowanie.
- Paradymaty: imperatywny, proceduralny, obiektowy
- Mechanizm wnioskowania typów.

Planowane cechy kompilatora:

- Wyjściowy kod zgodny z
- Możliwość używania bibliotek skompilowanych na platformę .NET.

1.2. Mechanizm wnioskowania typów. Przykłady w innych językach C#, F#, Scala.

1.3. generujący kod zgodny ze specyfikacją ECMA-335

3. Opis narzędzi

Język F#

4. Opis języka

Poniżej przedstawiono możliwości języka oraz zasady jego semantyki.

4.1. Moduły

Wszelki kod umieszczany jest w modułach. Każdy plik z kodem źródłowym tworzy moduł. Przestrzeń nazw dla modułu może zostać określona z użyciem słowa kluczowego module:

```
module Project::Component::ExampleModule
```

Deklaracja modułu jest opcjonalna. Jeżeli nie zostanie podana, nazwa wynikowego modułu zostanie określona na bazie nazwy pliku i jego lokalizacji.

4.2. Funkcje

Funkcje definiowane są z użyciem słowa kluczowego *fun.* Parametry funkcji oraz ich typy podawane są w nawiasach. Typ zwracany podawany jest po parametrach. Ciało funkcji zawiera się w nawiasach klamrowych. Przykład:

```
fun concat (a : int) (b : int) : string
{
    return a.ToString() + b.ToString();
}

fun main
{
    System::Console:.WriteLine(concat(1,2));
}
```

4.3. Moduły

Klasy deklarowane są z użyciem słowa kluczowego *class.* Członkowie klas zadeklarowani muszą być w następującej kolejności:

1. Deklaracje pól tylko do odczytu (*val*).
2. Deklaracje modyfikowalnych pól (*var*).
3. Konstruktory (*construct*).
4. Metody klasy (*fun*).

4.4. Typy

4.5. Instrukcje

4.6. Wyrażenia

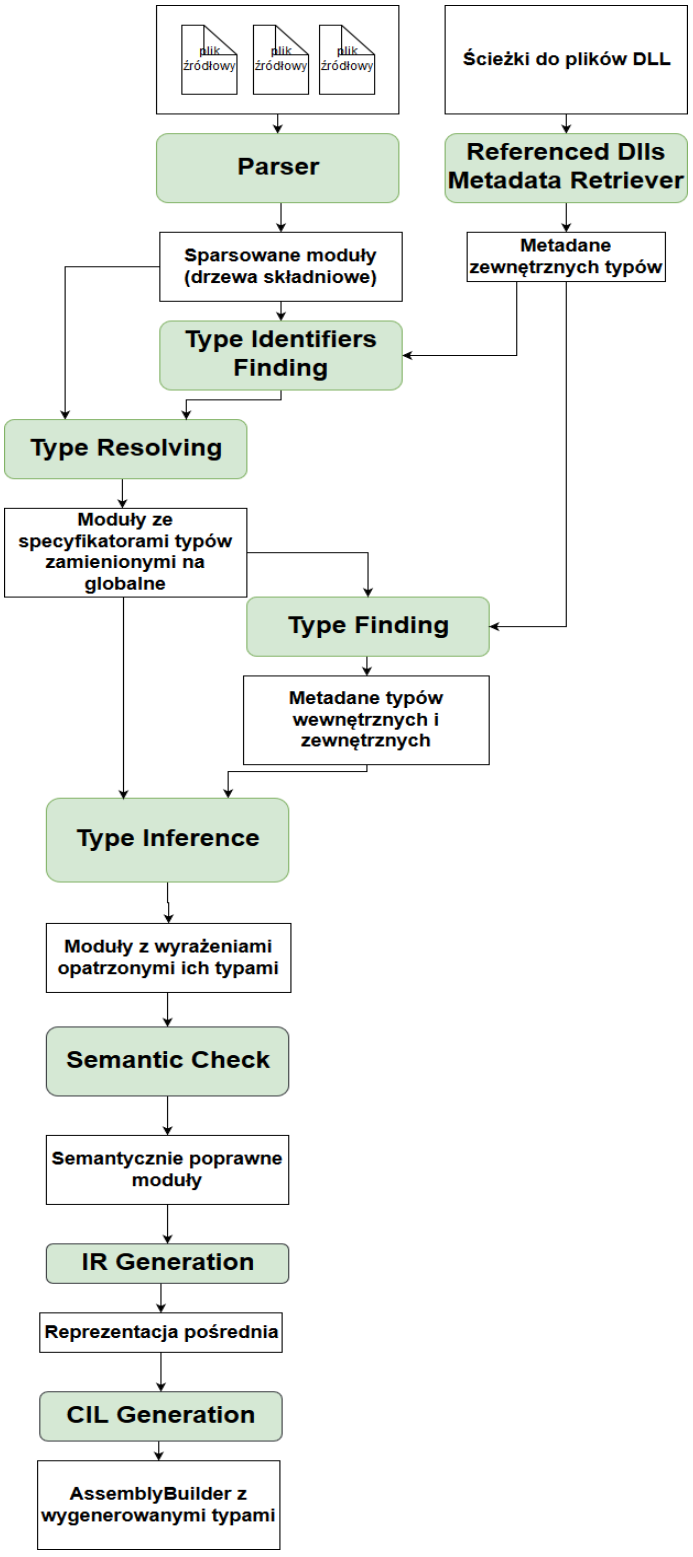
4.6.1. Wyrażenia binarne

= - przypisanie
|| – alternatywa logiczna
&& – koniunkcja logiczna
== - jest równy
!= - nierówny
<= - mniejszy lub równy

>= – większy lub równy
>, - większy
< - mniejszy
+ - plus
- - minus
* - mnożenie
/ - dzielenie
% - reszta z dzielenia
Wyrażenia unarne:
! – negacja logiczna
- - negacja arytmetyczna

5. Architektura kompilatora

5.1. Moduły kompilatora



6.

6.1. Rezultaty pośrednie kompilacji (Railway oriented programming).

6.2. Przetwarzanie AST. Katamorfizmy.

7. Parser

7.1. AST

7.2. Wybór sposobu implementacji parsera.

7.3. Zasada działania parsera typu top-down.

8. Rozwiązywanie typów.

8.1. Sprawdzanie czy typ jest zdefiniowany.

8.2. Podmienianie specyfikatorów typu na w pełni kwalifikowane.

9. Wnioskowanie typów.

9.1. Algorytm działania.

Funkcje poniżej mogą bazować na typach funkcji zdefiniowanych w tym samym module wcześniej.

Moduł wnioskowania typów wyznacza typ dla każdego wyrażenia w AST.

```
AstExpression = AstExpression of Expression<AstExpression>
```

Zamieniane jest na

```
InferredTypeExpression = InferredTypeExpression of Expression<InferredTypeExpression>
* TypeIdentifier
```

(do każdego sparsowanego wyrażenia dołączana jest informacja o typie - TypeIdentifier).

Proces odbywa się od dołu w górę drzewa pozwala to na ustalenie typów wyrażeń wykorzystujących wyrażenia wewnętrzne (wnioskowanie typów)

Informacja o typie używana jest w późniejszych etapach:

Sprawdzania semantyki - czy operandy instrukcji przypisania mają zgodne typy

Generatora reprezentacji pośredniej - nadanie typów wyrażeniom pośrednim

9.2. Algorytm znajdowania least-upper-bound.

10. Sprawdzanie semantyki.

10.1. Co jest sprawdzane.

- Czy zmienne lokalne oznaczone jako tylko do odczytu (modyfikator *val*) nie są nadpisywane po inicjalizacji.
- Czy pola oznaczone modyfikatorem *val* nie są nadpisywane poza konstruktorem.
- Czy wyrażenia wewnątrz instrukcji warunkowych *if* są typu *bool*.
- Jeżeli w deklaracji zmiennej został podany jej typ, to sprawdzane jest czy jest on zgodny z wywnioskowanym typem jej inicjalizatora.
- Jeżeli oczekiwanym wyjściem kompilacji jest plik .exe, to sprawdzane jest czy wśród modułów znajduje się dokładnie jedna funkcja o nazwie *main.*
- Operandy operatorów binarnych są tego samego typu.

11. Generowanie reprezentacji pośredniej.

12. Generowanie Common Intermediate Language.

Moduł ten odpowiada za wygenerowanie wyjściow.

```
let generateAssembly
        (assemblyBuilder : AssemblyBuilder)
        (referencedAssemblies : Assembly list)
        (modules : IR.Module list)
        (setEntryPoint : bool) =
```

13. Przykłady użycia kompilatora

1.Użycie

Plik wynikowy z kompilatorem - *Compile.*exe znajduje się w katalogu *release*. Zbudowanie całego projektu i wygenerowanie tego pliku jest też możliwe z użyciem skryptu w głównym katalogu - *build.cmd*.

W celu użycia kompilatora należy skopiować plik *Compile.exe* do folderu z kodem źródłowym.

Poniżej przedstawiono przykładowe programy napisane w stworzonym języku. Ich kod źródłowy można znaleźć w katalogu *samples*.

Przykład 1. Liczby pierwsze

```
fun main
{
    val primes = [2];

    var i = 3;
    while(i<100)
    {
        var isPrime = true;


        var finish = false;
        var j = 2;
        while(j*j < i && !finish)
        {
            if (i % j == 0)
            {
                isPrime = false;
                finish = true;
            }
            j = j + 1;
        }

        if(isPrime)
        {
            primes.Add(i);
        }
        i = i + 1;
    }
```

```
    val enumerator = primes.GetEnumerator();
    while(enumerator.MoveNext())
    {
        System::Console:.WriteLine(enumerator.Current);
    }

    enumerator.Dispose();
}
```

Przykład 2. Polimorfizm

```
module Classes

class Animal
{
    val _noise : string

    construct (noise : string)
    {
        _noise = noise;
    }

    fun MakeNoise
    {
        System::Console:.WriteLine(_noise);
    }
}

class Dog : Animal
{
    construct : ("Hau!")
    {
    }
}

class Cat : Animal
{
    construct : ("Miau!")
    {
    }
}

class Duck : Animal
{
    construct : ("")
    {
    }

    fun MakeNoise
    {
        System::Console:.WriteLine("Kwak!");
        System::Console:.WriteLine("Kwak!");
```

```
    }
}

fun createAnimal (animalType : string)
{
    if(animalType == "dog")
        return new Dog();
    else if(animalType == "cat")
        return new Cat();
    else
        return new Duck();
}

fun printAnimals (animals : System::Collections::Generic::List<Animal>)
{
    var i = 0;
    while(i < animals.Count)
    {
        val animal = animals.get_Item(i);
        System::Console:.Write("I am: ");
        System::Console:.WriteLine(animal.GetType());
        animal.MakeNoise();
        i = i + 1;
    }
}

fun main
{
    val dog = createAnimal("dog");
    val cat = createAnimal("cat");
    val duck = createAnimal("duck");

    val animals = [dog; cat; duck];
    printAnimals(animals);
}
```

Przykład 3. Użycie zewnętrznej biblioteki.

```
fun DrawFilledRectangle (x : int) (y : int)
{
    val bmp = new System::Drawing::Bitmap(x, y);
    val graphics = System::Drawing::Graphics:.FromImage(bmp);

    val imageRectangle = new System::Drawing::Rectangle(0, 0, x, y);
    graphics.FillRectangle(System::Drawing::Brushes:.White, imageRectangle);

    val blackBrush = new System::Drawing::SolidBrush(System::Drawing::Color:.Red);
    graphics.FillEllipse(blackBrush, imageRectangle);

    graphics.Dispose();
    return bmp;
}
```

```
fun main
{
    val bitmap = DrawFilledRectangle(640, 480);
    bitmap.Save("rectangle.jpg");
}
```

14. Podsumowanie.

14.1. Dalszy rozwój projektu.

14.2. W ramach dalszego rozwoju języka planowane jest dodanie nienulowalnych typów referencyjnych oraz wersja na .NET Standard (Linux).

15. Bibliografia