

# Formal Languages and Compilers

## Project

### roda Compiler

19598 Roland Bernard  
19615 Daniel Planötscher

16 giugno 2022

## 1 Introduzione

Roda è un piccolo linguaggio di programmazione stonly typed e compilato che utilizza una syntax moderna. Il linguaggio roda contiene le seguenti caratteristiche:

- Funzioni (con supporto per la chiamata di funzioni variadiche)
- Condizionali con espressioni if-else
- Cicli while
- Puntatori
- Espressioni aritmetiche su numeri interi e a virgola mobile
- Operazioni binarie su numeri interi
- Espressioni booleane
- Array
- Inference di tipi
- Alias di tipi

Il compilatore prova a dare messaggi di errore utili e informativi.

## 2 Linguaggio

Quello che segue è un semplice programma hello-world scritto in roda:

```
extern fn printf(fmt: *u8, ..);

pub fn main(): int {
    printf("Hello world!\n");
    return 0;
}
```

L'esempio seguente mostra le strutture di controllo supportate e il loro utilizzo:

```
extern fn printf(fmt: *u8, ..);

pub fn main(): int {
    let i = 1;
    while i <= 100 {
        if i % 3 == 0 && i % 5 == 0 {
            printf("FizzBuzz\n");
        } else if i % 3 == 0 {
```

```

        printf("Fizz\n");
    } else if i % 5 == 0 {
        printf("Buzz\n");
    } else {
        printf("%li\n", i);
    }
    i += 1;
}
return 0;
}

```

Le altre caratteristiche possono essere utilizzate nel modo seguente:

- Espressioni aritmetiche

```

x += 5;           // x = 8
// x += 0.5;      // real literals can not be implicitly converted to integers
let y: f64 = 0.5 + 0.1; // y = 0.6
// y += 5;        // integer literals can not be implicitly converted to floats

```

- Puntatori

```

let c = &a; // dereference a to store its value into c
&a = c;    // assign the value in c to the address pointed to by a

```

- Array

```

a[0] = 1; // assign the value 1 to the first element of a
a[1] = 2; // assign the value 2 to the second element of a
a[2] = a[1]; // assign the value 3 to the third element of a

```

- Inference di tipi

```

let b = 2.0; // b is a floating point number
let c = "Hello"; // c is a string
let d = true; // d is a boolean value
let e = 'a'; // e is a character
let f;
a += f; // f is also an integer

```

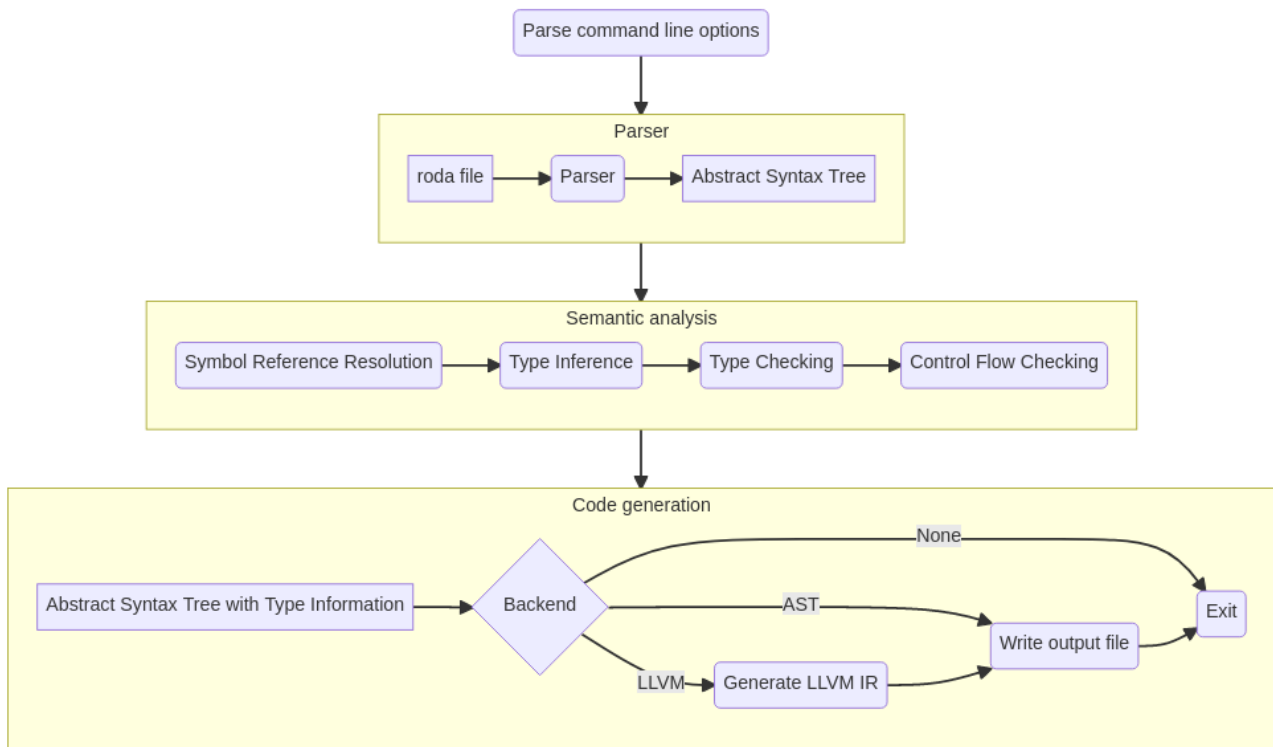
- Alias di tipi

```

type MyInt = int; // create a type alias for int
let a: MyInt = 1; // a is an integer

```

### 3 Implementazione



Abbiamo scelto un'architettura di compilazione multi-pass, che opera principalmente su un abstract syntax tree. Quello che segue sono i passaggi principali che compongono il processo di compilazione:

- **Parser:**

In questa fase, il compilatore legge il contenuto di un file e crea un abstract syntax tree utilizzando un lexer definito con Flex e un parser generato con Bison.

- **Symbol Reference resolution:**

In questa fase, il compilatore risolve tutti i riferimenti ai simboli, sia per le variabili che per i tipi, nell'abstract syntax tree. Questa fase genera e utilizza anche le symbol tables, creando un'entry di simbolo per ogni tipo, funzione e variabile definita. Questa fase genera e utilizza anche le symbol tables, creando un'entry di simbolo per ogni tipo, funzione e variabile definita. La tabella dei simboli è implementata come un tree con solo parent-pointer. Ogni nodo dell'albero rappresenta uno scope, implementato con hash tables. La ricerca dei simboli inizia da un nodo e prosegue fino alla root, fino a quando trova il simbolo. I tipi e i simboli condividono la stessa hash table, ma sono gestiti in modo diverso.

- **Type Inference:**

In questa fase, il compilatore infonde i tipi delle variabili e espressioni nell'abstract syntax tree. Questa fase risolve prima tutte le annotazioni di tipo esplicitamente fornite e i constraint impliciti di tipo (ad esempio, le nelle condizioni di if) e poi cerca d'inferire tutti gli altri tipi. L'inferenza dei tipi è implementata propagando i tipi usando l'attraversamento del grafo. Il compilatore inoltre, se il tipo non può essere altrimenti determinato, assume il tipo dei letterali interi e reali. Durante questa fase possono essere identificati e segnalati conflitti di tipo.

- **Type Checking:**

In questa fase, il compilatore controlla che tutti i tipi dell'abstract syntax tree siano corretti. Questo comprende, ad esempio, l'uso di solo tipi numerici per l'addizione o la moltiplicazione, o il controllo del numero corretto di argomenti che venga passato a una chiamata di funzione. Inoltre, questa fase si assicura che il tipo di ogni variabile ed espressione sia stato dedotto con successo. Al termine di questa fase, l'abstract syntax tree è stato popolato con tipi validi.

- **Control Flow Checking:**

In questa fase, il compilatore controlla che tutte le funzioni che dovrebbero restituire un valore, lo facciano effettivamente in ogni possibile branch attraverso il control-flow graph. Dopo questa fase, l'abstract syntax tree è garantito che rappresenti un programma valido, che può essere utilizzato per generazione di codice.

- **Code generation:**

In questa fase, il compilatore genera il codice per l'abstract syntax tree. Questa fase può sempre non fare nulla o scrivere l'albero della sintassi astratta in un file. Può anche, se compilato con il supporto per questo, generare l'IR LLVM e usarlo per generare un file di assembly o di object. Opzionalmente, questa fase può includere anche un'invocazione del compilatore C del sistema per il linking.

$Program \rightarrow RootStatements$   
 $RootStatements \rightarrow \epsilon$   
 $\quad | RootStatements RootStatement$   
 $\quad | RootStatements ;$   
 $RootStatement \rightarrow \text{pub fn Identifier ( Parameters ) OptionalType Block}$   
 $\quad | \text{extern fn Identifier ( Parameters ) OptionalType ;}$   
 $\quad | \text{fn Identifier ( Parameters ) OptionalType Block}$   
 $\quad | \text{type Identifier = Type ;}$   
 $OptionalType \rightarrow \epsilon$   
 $\quad | : Type$   
 $Parameters \rightarrow \epsilon$   
 $\quad | ParameterList$   
 $\quad | ParameterList ,$   
 $\quad | ParameterList , ..$   
 $ParameterList \rightarrow Parameter$   
 $\quad | ParameterList , Parameter$   
 $Parameter \rightarrow Identifier : Type$   
 $Block \rightarrow \{ Statements \}$   
 $Statements \rightarrow \epsilon$   
 $\quad | Statements BlockStatement$   
 $\quad | Statements ;$   
 $\quad | Statements Statement ;$   
 $BlockStatement \rightarrow \text{while Expression Block}$   
 $\quad | Block$   
 $\quad | If$   
 $If \rightarrow \text{if Expression Block}$   
 $\quad | \text{if Expression Block else Block}$   
 $\quad | \text{if Expression Block else If}$   
 $Statement \rightarrow Expression$   
 $\quad | Assignment$   
 $\quad | \text{return}$   
 $\quad | \text{return Expression}$   
 $\quad | \text{let Identifier OptionalType = Expression}$   
 $\quad | \text{let Identifier OptionalType}$   
 $Assignment \rightarrow Expression = Expression$   
 $\quad | Expression += Expression$   
 $\quad | Expression -= Expression$   
 $\quad | Expression *= Expression$   
 $\quad | Expression /= Expression$   
 $\quad | Expression \% = Expression$   
 $\quad | Expression >> = Expression$   
 $\quad | Expression << = Expression$   
 $\quad | Expression |= Expression$   
 $\quad | Expression \& = Expression$   
 $\quad | Expression \wedge = Expression$

$Type \rightarrow Identifier$   
 $\quad | ( )$   
 $\quad | * Type$   
 $\quad | [ Expression ] Type$   
 $Expression \rightarrow Identifier$   
 $\quad | Integer$   
 $\quad | Real$   
 $\quad | String$   
 $\quad | Boolean$   
 $\quad | ( )$   
 $\quad | ( Expression )$   
 $\quad | - Expression$   
 $\quad | + Expression$   
 $\quad | * Expression$   
 $\quad | \& Expression$   
 $\quad | ! Expression$   
 $\quad | Expression [ Expression ]$   
 $\quad | Expression ( Arguments )$   
 $\quad | Expression + Expression$   
 $\quad | Expression - Expression$   
 $\quad | Expression * Expression$   
 $\quad | Expression / Expression$   
 $\quad | Expression \% Expression$   
 $\quad | Expression \& Expression$   
 $\quad | Expression | Expression$   
 $\quad | Expression \wedge Expression$   
 $\quad | Expression \&\& Expression$   
 $\quad | Expression || Expression$   
 $\quad | Expression >> Expression$   
 $\quad | Expression << Expression$   
 $\quad | Expression == Expression$   
 $\quad | Expression != Expression$   
 $\quad | Expression <= Expression$   
 $\quad | Expression >= Expression$   
 $\quad | Expression > Expression$   
 $\quad | Expression < Expression$   
 $Integer \rightarrow \text{INT}$   
 $\quad | \text{CHAR}$   
 $Real \rightarrow \text{REAL}$   
 $String \rightarrow \text{STR}$   
 $Boolean \rightarrow \text{true}$   
 $\quad | \text{false}$   
 $Identifier \rightarrow \text{ID}$   
 $Arguments \rightarrow \epsilon$   
 $\quad | ArgumentList$   
 $\quad | ArgumentList ,$   
 $ArgumentList \rightarrow Expression$   
 $\quad | ArgumentList , Expression$