Formal Languages and Compilers
Project
roda Compiler

19598 Roland Bernard
19615 Daniel Planötscher

17 giugno 2022

1 Introduzione

Roda è un piccolo linguaggio di programmazione strongly 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

Inoltre, 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

```
let x = 1 + 2;          // x = 3
x += 5;                 // x = 8
// x += 0.5;            // real literals can not be implicitly converted to integers
x += 0.5 as int;        // x = 8
let y: f64 = 0.5 + 0.1; // y = 0.6
// y += 5;              // integer literals can not be implicitly converted to floats
y += 5 as f64;          // y = 5.6
```

- Puntatori

```
let a = *b; // take the address of b and store it into a
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

```
let a: [3]int; // create an array of 3 ints
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 a = 1;                  // a is an integer
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
let g = [[1, 2], [3, 4]];   // g is of type [2][2]int;
```

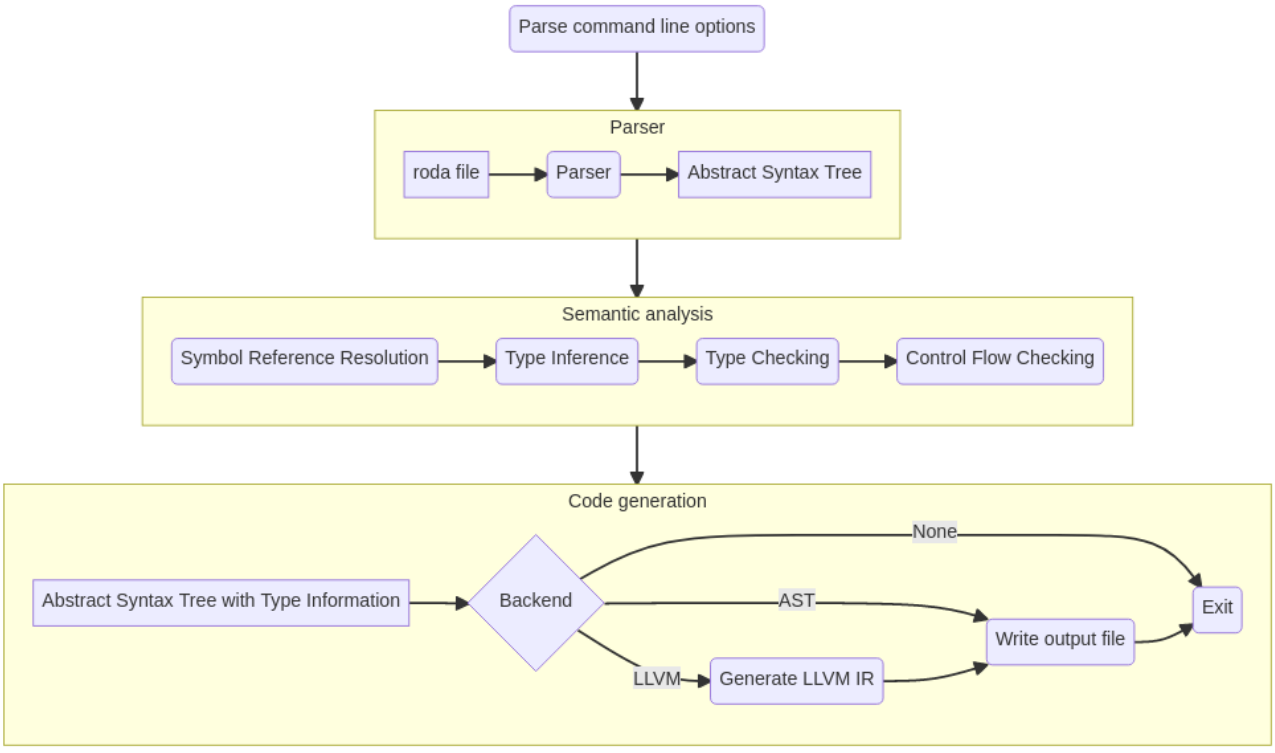
- Alias di tipi

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

Operatori	Associatività
Function calls, array indexing	
Unary `+ - * ! &`	
`as`	
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`== != < > <= >=`	left to right
`&&`	left to right
`||`	left to right

La tabella precedente mostra la precedenza degli operatori del linguaggio, con gli operatori a precedenza più alta indicati in alto. La grammatica della lingua viene mostrata nell'ultima pagina ed è una grammatica ambigua che, con l'uso delle precedenze di sopra, può essere resa non ambigua.

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 una hash table. 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 (infers) i tipi delle variabili e delle 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 "if"). 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 → *RootStatements*

RootStatements → ϵ
| *RootStatements RootStatement*
| *RootStatements* `;`

RootStatement → `pub fn` *Identifier* `(` *Parameters* `)` *OptionalType Block*
| `extern fn` *Identifier* `(` *Parameters* `)` *OptionalType* `;`
| `fn` *Identifier* `(` *Parameters* `)` *OptionalType Block*
| `type` *Identifier* `=` *Type* `;`

OptionalType → ϵ
| `:` *Type*

Parameters → ϵ
| *ParameterList*
| *ParameterList* `,`
| *ParameterList* `, ..`

ParameterList → *Parameter*
| *ParameterList* `,` *Parameter*

Parameter → *Identifier* `:` *Type*

Block → `{` *Statements* `}`

Statements → ϵ
| *Statements BlockStatement*
| *Statements* `;`
| *Statements Statement* `;`

BlockStatement → `while` *Expression Block*
| *Block*
| *If*

If → `if` *Expression Block*
| `if` *Expression Block* `else` *Block*
| `if` *Expression Block* `else` *If*

Statement → *Expression*
| *Assignment*
| `return`
| `return` *Expression*
| `let` *Identifier OptionalType* `=` *Expression*
| `let` *Identifier OptionalType*

Assignment → *Expression* `=` *Expression*
| *Expression* `+=` *Expression*
| *Expression* `-=` *Expression*
| *Expression* `*=` *Expression*
| *Expression* `/=` *Expression*
| *Expression* `%=` *Expression*
| *Expression* `>>=` *Expression*
| *Expression* `<<=` *Expression*
| *Expression* `|=` *Expression*
| *Expression* `&=` *Expression*
| *Expression* `^=` *Expression*

Integer → INT
| CHAR

Real → REAL

String → STR

Boolean → `true`
| `false`

Identifier → ID

Type → *Identifier*
| `( )`
| `*` *Type*
| `[` *Expression* `]` *Type*
| `fn (` *Types* `)`
| `fn (` *Types* `) :` *Type*

Types → ϵ
| *TypeList*
| *TypeList* `,`
| *TypeList* `, ..`

TypeList → *Type*
| *TypeList* `,` *Type*

Expression → *Identifier*
| *Integer*
| *Real*
| *String*
| *Boolean*
| `sizeof` *Type*
| `( )`
| `[` *List* `]`
| `(` *Expression* `)`
| `-` *Expression*
| `+` *Expression*
| `*` *Expression*
| `&` *Expression*
| `!` *Expression*
| *Expression* `as` *Type*
| *Expression* `[` *Expression* `]`
| *Expression* `(` *List* `)`
| *Expression* `+` *Expression*
| *Expression* `-` *Expression*
| *Expression* `*` *Expression*
| *Expression* `/` *Expression*
| *Expression* `%` *Expression*
| *Expression* `&` *Expression*
| *Expression* `|` *Expression*
| *Expression* `^` *Expression*
| *Expression* `&&` *Expression*
| *Expression* `||` *Expression*
| *Expression* `>>` *Expression*
| *Expression* `<<` *Expression*
| *Expression* `==` *Expression*
| *Expression* `!=` *Expression*
| *Expression* `<=` *Expression*
| *Expression* `>=` *Expression*
| *Expression* `>` *Expression*
| *Expression* `<` *Expression*

List → ϵ
| *ListNonEmpty*
| *ListNonEmpty* `,`

ListNonEmpty → *Expression*
| *ListNonEmpty* `,` *Expression*