Chora

Di seguito sono forniti alcuni dettagli relativi al funzionamento del compilatore e ai costrutti presenti nel linguaggio. Si tenga presente che nell'implementazione del compilatore sono presenti ancora dei bug, pertanto è probabile che durante la compilazione si verifichino degli errori. Ad esempio, se viene dato in input un programma non sintatticamente corretto, il compilatore potrebbe entrare in un loop infinito.

Esecuzione

Requisiti e installazione

Poichè come backend è utilizzato LLVM, è necessario averlo installato sul proprio computer (su Ubuntu `sudo apt-get install llvm`). Durante la fase di linking è inoltre necessario disporre della standard library del C++ (su cui dipende LLVM).

Di seguito, il comando per creare l'eseguibile (salvato nella sottocartella `bin` del progetto).

```
gcc -std=c11 -w `llvm-config --cflags` \
    src/main.c src/lexer/*.c src/parser/*.c src/llvm-ir/*.c src/error/*.c
src/file/*.c src/ast/*.c src/std/*.c \
    `llvm-config --ldflags --libs core analysis native --system-libs` -
lstdc++ -o bin/chc
```

In alternativa, si può usare CMake:

```
cmake .
make
```

Compilazione

Per compilare un file (che deve avere estension `.ch`) è sufficiente eseguire il seguente comando:

```
chc file.ch
```

Se il programma è valido, viene generato un file `file.ll` contenente il corrispondente codice intermedio. Per generare un eseguibile, vi sono due possibilità:

1. clang

```
clang file.ll
```

2. llc + gcc/clang

```
llc file.ll
gcc file.s
```

Compilatore

Implementazione

Attualmente, il frontend del compilatore consiste delle seguenti quattro fasi:

1. analisi lessicale
2. analisi sintattica
3. analisi semantica
4. generazione del codice intermedio

Per semplificare l'implementazione, le fasi intermedie, ossia quelle di analisi sintattica e semantica, sono svolte contemporaneamente. In altre parole durante la costruzione del parse tree è eseguita anche l'analisi semantica. In questo modo, se da un lato è possibile fare un solo pass dell'AST, dall'altro lato le funzionalità implementate dal compilatore (e, quindi, offerte dal linguaggio) sono più ridotte. Ad esempio, prima di poter fare riferimento a una variabile/funzione è necessario averla precedentemente definita, in quanto il compilatore per poter valutare un'espressione deve 'conoscere' tutti gli elementi di cui è costituita.

Anche la type inference è abbastanza limitata. Si prenda il seguente esempio:

```
fn main(): u8 {

    let x = 100; // necessario specificare il tipo let x = 100u8;
    x
}
```

Poichè il compilatore esamina ogni statement/espressione una sola volta, quando analizza la dichiarazione della variabile `x` assegna ad essa il tipo `i32`, causando un errore quando tale variabile viene usata nell'espressione di ritorno, che deve essere di tipo `u8` (si noti che Chora, come Rust, non effettua alcuna conversione implicita). Se si eseguissero più pass dell'albero sarebbe possibile dedurre che il tipo di `x` deve essere `u8`. Le stesse limitazioni si hanno quando si specificano i valori dei parametri nell'invocazione di una funzione.

Per quanto riguarda la fase di analisi sintattica, essa è implementata utilizzando un predictive recursive descent parser (con un lookahead pari a 2) per il parsing degli item e degli statetement, mentre per il parsing delle espressioni è utilizzato un Pratt parser (basato sulla seguente implementazione). Come detto, durante la fase di parsing viene generato un abstract syntax tree (misto a un parse tree per migliorare la stampa degli errori) e viene inoltre creata una symbol table implementata tramite una lista di hash table.

La fase di generazione del codice intermedio è implementata facendo uso della C API Interface esposta da LLVM. Attualmente, tutte le ottimizzazioni sono effettuate nella fase di backend.

Specifica

Di seguito è fornita una specifica della grammatica (in extended BNF).

Identificatori

```
NAME_ID ::=
      _ ( LETTER | DEC_DIGIT | _ )+
    | LETTER ( LETTER | DEC_DIGIT | _ )*

LETTER ::= [a-zA-Z]

OP_ID ::= (OP_CHAR)+

OP_CHAR ::= ! | % | ^ | & | * | - | + | = | : | @ | ~ | # | '|' | \ | < | >
| / | ?
```

Tipi

```
Type ::=
      ParenthesizedType
    | InferredType
    | IdentifierType
    | EmptyType
    | NeverType

ParenthesizedType ::= ( Type )

InferredType ::= _

IdentifierType ::= NAME_ID

EmptyType ::= ()

NeverType ::= !
```

Attualmente, in Chora sono presenti soltanto 5 tipi di dato:

1. il never type `!`, ossia il tipo di dato restituito da tutte le istruzioni che interrompono l'esecuzione sequenziale per saltare a una nuova istruzione (è quindi analogo a `!` presente in Rust e a `Nothing` in Scala)
2. l'emtpy type `()` (analogo al tipo `void` in C/C++, `()` in Rust e `Unit` in Kotlin/Scala)
3. i tipi di dato intero, che possono essere signed o unsigned e sono, rispettivamente:
    - `u8`, `u16`, `u32`, `u64` (a differenza di Rust non vi è `usize`)
    - `i8`, `i16`, `i32`, `i64` (a differenza di Rust non vi è `isize`)
4. i tipi floating point, che possono essere `f32` e `f64`

5. il tipo booleano, che può assumenre i valori `true` e `false`

La notazione di un literal intero è analoga a quella usata in Rust o C++. Come in Rust, qualora un integer suffix non sia specificato (e non possa essere dedotto da altre informazioni, come il tipo della variabile), a un intero è assegnato automaticamente il tipo `i32`.

```
INTEGER_LITERAL ::= ( BIN_LITERAL | OCTAL_LITERAL | DEC_LITERAL |
HEX_LITERAL ) ( _ )* (INTEGER_SUFFIX)?

BIN_LITERAL ::= 0b ( BIN_DIGIT | ' )*

OCTAL_LITERAL ::= 0o ( OCTAL_DIGIT | ' )*

DEC_LITERAL ::= DEC_DIGIT ( DEC_DIGIT  | ' )*

HEX_LITERAL ::= 0x ( HEX_DIGIT | ' )*

BIN_DIGIT ::= [0-1]

OCTAL_DIGIT ::= [0-7]

DEC_DIGIT ::= [0-9]

HEX_DIGIT ::= [0-9a-fA-F]

INTEGER_SUFFIX ::= u8 | u16 | u32 | u64 | i8 | i16 | i32 | i64
```

Anche la notazione dei floating point literals è simile a quella in C++ e Rust, ma in Chora è possibile definire floating points anche in formato binario, ottale e decimale. Anche in questo caso, se non è specificato un suffisso (o se non è possibile ricavarlo dal contesto), è automaticamente assegnato il tipo `f64`.

```
FLOAT_LITERAL ::= ( BIN_FLOAT | OCTAL_FLOAT | DEC_FLOAT | HEX_FLOAT ) ( _
)* (FLOAT_SUFFIX)?

BIN_FLOAT ::= 0b ( BIN_DIGIT | ' )* (. ( BIN_DIGIT | ' )* )? ( (e | E) ( +
| - )? ( BIN_DIGIT | ' )* )?

OCTAL_FLOAT ::= 0o ( OCTAL_DIGIT | ' )* (. ( OCTAL_DIGIT | ' )* )? ( (e |
E) ( + | - )? ( OCTAL_DIGIT | ' )* )?

DEC_FLOAT ::= DEC_DIGIT ( DEC_DIGIT | ' )* (. ( DEC_DIGIT | ' )* )? ( (e |
E) ( + | - )? ( DEC_DIGIT | ' )* )?

HEX_FLOAT ::= 0o ( HEX_DIGIT | ' )* (. ( HEX_DIGIT | ' )* )? ( (e | E) ( +
| - )? ( HEX_DIGIT | ' )* )?

INTEGER_SUFFIX ::= f32 | f64
```

File

```
ChoraFile ::= (Item)+

Item ::=
      FunctionItem
    | ConstantItem
    | StaticItem
```

In Chora, in ogni translation unit è possibile definire (a livello globale) item di 3 differenti tipoligie:

1. costanti
2. variabili statiche
3. funzioni

Dato che, attualmente, ogni progetto consiste di una sola translation unit, quest'ultima deve contenere una funzione `main()` da cui far iniziare l'esecuzione. Un'altra limitazione è che all'interno di un item non è possibile fare riferimento ad altri item se non sono stati ancora dichiarati. Ad esempio, nel corpo di una funzione non è possibile invocare una funzione che non è stata ancora definita (non è quindi possibile definire funzioni mutualmente ricorsive).

Funzioni

```
FunctionItem ::= fn NAME_ID ( Parameters? ) (ReturnType)? BlockExpression

Parameters ::= Param ( , Param )* (,)?

Param ::= Pattern : Type ( = Expression )?

ReturnType ::= : Type
```

La sintassi delle funzioni in Chora è ispirata a quella in Rust, Kotlin e Scala. Come in questi linguaggi, inoltre, qualora il tipo di ritorno non sia specificato, alla funzione viene assegnato il tipo `()`. Come in Scala (e Kotlin), è possibile fare l'overloading delle funzioni, ossia è possibile definire funzioni con lo stesso nome, ma parametri differenti e/o diverso tipo di ritorno. È inoltre possibile definire valori di default per i parametri. Nell'espressione usata per definire un valore di default è possibile fare riferimento ai parametri precedenti, ma non a quelli successivi e non è possibile fare riferimento alla funzione stessa (che è invece possibile in Scala).

Costanti

```
ConstantItem ::= const Pattern ( : Type )? = Expression ;
```

Le costanti in Chora sono analoghe a quelle in Rust, ossia sono valori (costanti) a cui è possibile assegnare un nome e a cui non corrisponde alcuna locazione in memoria. In altri termini, tali valori sono copiati in ogni

punto in cui si fa riferimento a essi. Attualmente, è possibile assegnare alle costanti soltanto dei literal e non il risultanto di espressioni più complesse (come è invece possibile con `constexpr` in C++ o con `const` in Rust). Ovviamente, non è possibile definire una costante con un mutable pattern.

Variabili statiche

```
StaticItem ::= static Pattern ( : Type )? = Expression ;
```

Le variabili statiche in Chora sono invece analoghe a quelle in C++. A ogni variabile statica è assegnata una locazione in memoria che persiste durante l'intera esecuzione del programma (nota: come spiegato in seguito, tale affermazione non è vera qualora si usi un wildcard pattern).

A differenza delle costanti, il cui valore è calcolato a tempo di compilazione, il valore di una variabile statica globale è calcolato durante la fase di startup del programma, pertanto è possibile assegnare il risultato di un'espressione più complessa (non valutabile a compile time). Le variabili statiche definite all'interno di un blocco sono invece inizializzate soltanto la prima volta in cui l'esecuzione del programma passa per la loro dichiarazione. In altri termini, se una variabile statica è definita all'interno di un blocco che non viene mai eseguito, tale variabile non verrà mai inizializzata.

Statements

```
Statement ::=
      ;
    | Item // no FunctionItem
    | LetStatement
    | ExpressionStatement

LetStatement ::= let Pattern ( : Type )? = Expression ;

ExpressionStatement ::=
      ExpressionWithoutBlock ;
    | ExpressionWithBlock (;)?
```

Espressioni

```
Expression ::=
      ExpressionWithBlock
    | ExpressionWithoutBlock

ExpressionWithoutBlock ::=
      BreakExpression | ReturnExpression
    | AssignmentExpression
    | InfixExpression
    | PrefixExpression
    | PostfixExpression
    | FunctionCallExpression | LiteralExpression | VariableExpression |
```

```
GroupedExpression

ExpressionWithBlock ::=
      BlockExpression
    | IfExpression
    | LoopExpression
    | WhileExpression
```

A differenza degli statement, che non hanno valore di ritorno, tutte le espressioni hanno un valore di ritorno. Per semplicità, è qui mostrata una grammatica ambigua che non tiene conto della precedenza e dell'associatività delle diverse tipologie di espressione. Per quanto riguarda le `ExpressionWithoutBlock`, esse sono elencate per gradi crescenti di precedenza.

Espressioni 'con blocco'

```
Label ::= @ NAME_ID

BlockExpression ::= ( Label : )? { (Statement)* (Expression)? }

IfExpression ::= if Expression BlockExpression ( else ( BlockExpression |
IfExpression ) )?

LoopExpression ::= loop BlockExpression

WhileExpression ::= while Expression BlockExpression
```

In Chora, un `block` è uno dei costrutti principali. Come in Rust, gli statement in un blocco sono eseguiti sequenzialmente e viene poi valutata l'espressione finale, il cui valore è restituito. Un blocco introduce inoltre un nuovo livello di scope anonimo, ossia tutti gli item dichiarati all'interno di un blocco sono visibili soltanto all'interno di esso (e dei suoi figli). Una differenza rispetto a Rust (e ad altri linguaggi di programmazione), è che in Chora è possibile assegnare dei nomi (label) ai blocchi. Tale label può essere poi usata in una break o return expression per restituire un valore dal blocco corrispondente. Per evitare di dover assegnare un nome a tutti i blocchi, ai blocchi associati a una `if clause`, a un `loop`, a un `while` e a una funzione sono associati nomi secondari (rispettivamente `if/else`, `loop`, `while` e il nome della funzione).

Espressioni 'senza blocco'

```
BreakExpression ::= break (Label)? (Expression)?

ReturnExpression ::= return (Label)? (Expression)?

AssignmentExpression ::= Expression = Expression

InfixExpression ::= Expression OP_ID Expression

PrefixExpression ::= OP_ID Expression
```

```
PosfixExpression ::= Expression OP_ID

FunctionCallExpression ::= NAME_ID ( (Arguments)? )

Arguments ::= Argument ( , Argument )* (,)?

Argument ::= ( NAME_ID = )? Expression

LiteralExpression ::=
      INTEGER_LITERAL
    | FLOAT_LITERAL
    | BOOL_LITERAL

VariableExpression ::= NAME_ID

GroupedExpression ::= ( Expression )
```

Come accennato prima, una peculiarità di Chora riguarda le espressioni `return` e `break`. Mentre nella maggior parte dei linguaggi di programmazione esse sono usate rispettivamente per interrompere l'esecuzione di una funzione e per interrompere l'esecuzione di un loop, in Chora, ambedue sono usate per interrompere l'esecuzione di un blocco ed eventualmente restituire un valore. Come detto prima, per specificare quale blocco interrompere è possibile usare una label. La differenza tra `return` e `break` riguarda la ricerca del blocco. `return` effettua la ricerca a partire dal blocco più esterno (che è sempre quello di una funzione), che è quello selezionato in caso in cui nessuna label sia specificata. `break`, invece, effettua la ricerca a partire dal blocco più interno.

Un'altra caratteristica di Chora è che non vi sono veri e propri operatori, eccetto per l'operatore di assegnamento `=` e l'operatore di conversione `as`. Come in Scala, infatti, tutti gli altri operatori sono metodi che ogni tipo implementa. Attualmente, si possono usare soltanto gli operatori implementati direttamente dal compilatore, in quanto non è possibile definire nuovi metodi. A differenza di Scala (e di Swift), in Chora vi sono soltanto i seguenti gradi di precedenza:

1. `as` ha la precedenza più alta
2. postfix operators
3. prefix operators
4. infix operators
5. `=`

Come si può vedere, quindi, tutti gli operatori della stessa tipologia hanno la stessa precedenza e le espressioni sono valutate da sinistra a destra (come in Smalltalk). Pertanto, un'espressione del tipo `a + b * c` è valutata come `(a + b) * c` e non come `a + (b * c)`.

Un'altra somiglianza con Scala (e Kotlin) riguarda le chiamate di funzioni. È infatti possibile passare gli argomenti in un ordine differente da quello in cui i parametri sono dichiarati. È sufficiente specificare il nome del parametro a cui si fa riferimento. È ovviamente possibile non specificare un valore per i parametri a cui è associato un valore di default.

Pattern

```
Pattern ::=
      IdentifierPattern
    | WildcardPattern

IdentifierPattern ::= (mut)? NAME_ID

WildcardPattern ::= _
```

Come accennato precedentemente, qualora il nome di una variabile sia `_`, a essa non è allocata nessuna area in memoria. L'espressione di inizializzazione è comunque valutata.

Esempi

Per alcuni esempi di programmi in Chora, si vedano i file nella subdirectory `examples`.