

Cinc

Linguaggio ed Implementazione

Claudia Strazzari

Fabio G. Strozzi

Matteo Benevelli

<http://www.cs.unibo.it/~fstrozzi/cinc.html>

Prefazione

Cinc è un linguaggio di programmazione imperativo, fortemente tipato, in parte compilato ed in parte interpretato. Del compilatore abbiamo sviluppato il front-end, mentre la traduzione del codice intermedio è stata lasciata a Jasmin, un back-end che produce byte-code Java; pertanto, il codice prodotto può essere interpretato perfettamente dalla Java Virtual Machine.

Cinc fornisce semplici costrutti per il controllo di flusso, i tipi primitivi essenziali (interi e booleani), e due tipi strutturati (array e stringhe).

Al momento della progettazione abbiamo cercato di ispirarci ad una sintassi facile da memorizzare ed intuitiva; l'intento era quello di prendere spunto dai vari linguaggi che conoscevamo per raccogliere i costrutti più interessanti; per esempio, dal C e da Java sono stati presi gli assegnamenti, le espressioni ed i tipi; da Python l'ispirazione per la struttura dei blocchi, mentre dal formalismo dell'analisi matematica le dichiarazioni delle funzioni. Il nome, ironicamente, sottolinea la differenza tra Cinc e un linguaggio come il C, che comunque è stato un importante strumento di apprendimento.

I miglioramenti riguardano:

- il controllo dei tipi;
- la possibilità di istanziare variabili ovunque nel codice;
- la netta separazione tra espressioni e comandi (per esempio non si possono fare assegnamenti all'interno della guardia);
- la mancanza del comando goto (come prevede la programmazione strutturata);
- i commenti annidati.

Ovviamente, molto ha giovato l'utilizzo della Java Virtual Machine che ha fornito il supporto run-time per un potente meccanismo di garbage collection.

Cinc è nato per uno scopo didattico; i tempi ristretti non hanno permesso di progettare costrutti per la creazione di tipi da parte dell'utente (record, struct o union), né regole di type checking a run-time.

1 Il Linguaggio

1.1 Principi fondamentali

Ogni programma scritto in Cinc è composto da due parti:

- una parte di dichiarazioni di variabili globali (opzionale);
- una parte di definizione di funzioni.

Lo scope, in Cinc, è statico; la visibilità degli identificatori è determinabile a tempo di compilazione e non dipende dalla sequenza di computazioni a run-time. Le funzioni sono definite tutte allo stesso livello e non esistono dichiarazioni annidate (vedi Pascal), pertanto, sono ovunque visibili nel codice.

Non è necessario che funzioni e variabili abbiano nomi diversi; il contesto in cui sono richiamati gli identificatori è sufficiente per distinguerne il tipo. È necessario, invece, che nel codice compaia una funzione principale (`main`), che rappresenta l'inizio dell'esecuzione.

1.2 Dichiarazioni

Per istanziare delle variabili si utilizza una sintassi molto simile a Java; la dichiarazione inizia con la scelta del tipo seguito da una lista di identificatori separati da virgole. Non sono ammessi assegnamenti durante la dichiarazione. I tipi del linguaggio sono:

- interi (`int`);
- booleani (`bool`);
- stringhe di caratteri (`string`);
- array.

Gli array sono definiti per composizione sui tre tipi basilari (interi, booleani e stringhe). Possono essere multidimensionali e vengono allocati dinamicamente (nell'heap) grazie all'istruzione `new`. La dichiarazione è analoga a quella di Java. Per esempio:

```
int[] [] a, b, c;  
a = new int[5][5];
```

dichiara tre array di interi (`a`, `b` e `c`) e successivamente alloca dinamicamente lo spazio di `a`.

Una volta dichiarate, le variabili possono essere utilizzate immediatamente; Cinc si incarica di inizializzarle con un valore di default al momento della istanziazione. A differenza del C, possono essere dichiarate ovunque nel corpo delle funzioni.

1.3 Funzioni

Un aspetto sul quale fin da subito abbiamo lavorato è stato la scelta della sintassi per la definizione delle funzioni. Linguaggi come C o C++ permettono all'utente di dichiarare i prototipi delle funzioni. Indubbiamente, una scelta di questo tipo avrebbe semplificato molto il type-checking, soprattutto nei casi di mutua ricorsione. Alla fine però, si è optato per non utilizzare prototipi, in modo che il linguaggio risultasse più snello nella stesura, senza perdita di leggibilità. La definizione di una funzione consiste in una intestazione che ne stabilisce il nome e il tipo (parametri formali e tipo di ritorno), seguita dal corpo della funzione stessa. In conseguenza a questa scelta, è risultato necessario permettere l'invocazione di funzioni non ancora dichiarate; Cinc tiene traccia di queste chiamate per effettuare i controlli di tipo non appena incontra le rispettive definizioni. Ovviamente, se anche solo una delle funzioni invocate non viene definita, Cinc comunica un errore.

Una funzione che non ha parametri formali ha come tipo di input `void`. Una funzione che non restituisce valori al chiamante deve dichiarare il tipo di ritorno come `void`; in questo caso verranno considerate errate le istruzioni `return` nel corpo della funzione.

Cinc, basandosi su Java, utilizza un *value model* per le variabili di tipo primitivo ed un *reference model* per quelle strutturate (array e stringhe). Il passaggio di parametri alle funzioni avviene sempre per valore, anche quando ad essere copiato è un riferimento (ad array o stringhe).

1.4 Espressioni

In Cinc espressioni e comandi sono componenti distinti del linguaggio e non possono essere intercambiati.

Per permettere l'utilizzo di funzioni non ancora definite all'interno di espressioni, Cinc vanta di un piccolo sistema di inferenza dei tipi (descritto nel capitolo 2.3). Un problema che si è posto, infatti, era stabilire quale fosse il tipo di una espressione derivata dalla chiamata ad una funzione non ancora definita. Non conoscendo il tipo di ritorno di questa funzione, abbiamo optato per ritornare un tipo generico (`star`) che possa essere unificato senza problemi con qualsiasi altro tipo (una sorta di tipo polimorfo). Per poter proseguire nel type checking e fare un'analisi precisa delle espressioni, era necessario unificare i tipi `star` prima possibile. Ciò è stato ottenuto vietando l'overloading degli operatori. Per esempio, l'operatore '+' lavora solo su interi e non può essere utilizzato per concatenare le stringhe (per tale operazione esiste l'operatore '^'). In questo modo, è sempre determinato il tipo di una espressione (vedi Tabella 1).

Le espressioni booleane vengono calcolate tramite il *short circuit*; quando il risultato è noto l'espressione non deve essere calcolata per intero.

Operatori	Tipo degli Operandi	Tipo dell'Espressione
\wedge	string	string
$+$, $-$, $*$, $/$, exp , mod , abs	int	int
$==$, $<=$, $>=$, $<$, $>$, $!=$	int	bool
or , not , and	bool	bool

Tabella 1: Operatori, tipi degli operandi e delle espressioni

2 L'implementazione

2.1 Analisi lessicale

L'analizzatore lessicale (scanner) è stato ottenuto col supporto di Flex.

Durante questa prima fase viene aggiornata una struttura dati (`yyvaloc`) contenente numero di riga e di colonna di inizio e di fine di ogni token. Grazie a questa informazione l'emissione degli errori può risultare molto precisa perchè permette di identificare porzioni specifiche di codice (e non semplicemente i numeri di riga).

I commenti annidati sono riconosciuti dallo scanner grazie all'utilizzo delle *start condition*; quando Flex riconosce un commento attiva le regole specifiche della start condition *comment* e aggiunge lo stato su uno stack. Ad ogni chiusura di commento verrà tolto uno stato dalla pila fino a quando non si ritornerà allo stato iniziale; a questo punto l'analisi lessicale ripartirà con le regole normali. Le start condition sono state utilizzate anche per individuare le costanti di tipo stringa.

2.2 Analisi sintattica e tabella dei simboli

Il parser di Cinc è stato generato in modo automatico utilizzando Bison. Al riconoscimento delle produzioni vengono svolti tutti i controlli semantici e si procede alla costruzione dell'albero sintattico astratto. Lo strumento essenziale in questa fase è la tabella dei simboli: essa associa ad ogni identificatore una serie di caratteristiche (tra cui il tipo per le variabili).

Abbiamo scelto di utilizzare due tabelle distinte per variabili e funzioni con l'intento di velocizzare le ricerche e semplificare la rappresentazione interna. La struttura di base di entrambe è costituita da una tabella hash per il recupero degli identificatori e una pila di strutture contenenti le informazioni relative agli identificatori. La funzione hash, utilizzata per l'indicizzazione degli elementi della tabella dei simboli, applica il metodo della moltiplicazione, per cui la dimensione dell'array non risulta un parametro critico.

Per rappresentare l'annidamento degli ambienti, rispettando le regole di scoping, è stata introdotta una pila di strutture; in testa troviamo l'ultimo ambiente attivato e un puntatore alla lista delle variabili visibili. L'inizio di un nuovo blocco causa l'inserimento di un elemento in testa alla pila e, ad ogni dichiarazione di variabile, segue l'introduzione di un elemento nella pila delle variabili dichiarate. Le funzioni non necessitano della gestione di ambienti perché vengono dichiarate tutte allo stesso livello (ambiente globale) e sono perciò visibili ovunque.

Gli elementi all'interno delle tabelle dei simboli vengono memorizzati in una struttura contenente:

- stringa identificativa;
- locazione della definizione;
- alcuni flag indispensabili per il controllo dei tipi;
- puntatore ad una struttura dati contenente informazioni sul tipo;

Per la memorizzazione di queste strutture si è scelto di utilizzare una pila, percorribile attraverso una coppia di puntatori, a seconda dell'informazione che se ne vuole ricavare. Con l'utilizzo di un'unica struttura, viene mantenuto perciò l'elenco delle variabili dichiarate all'interno di un blocco ma anche la pila delle variabili dichiarate con uguale stringa identificativa (la testa rappresenterà dunque quella visibile). L'uscita da un blocco causa la rimozione dell'ambiente in cima alla pila e di tutte le variabili in esso dichiarate.

Il risultato è una struttura che permette l'allocazione completamente dinamica dello spazio necessario alla memorizzazione delle variabili, senza peraltro perdite in termini di prestazioni.

2.3 Analisi semantica e deduzione dei tipi

La componente fondamentale dell'analisi semantica in un linguaggio fortemente tipato è il controllo dei tipi. Cinc effettua tutti i controlli durante il parsing del codice e contemporaneamente alla generazione dell'albero sintattico astratto. Il problema principale che abbiamo dovuto affrontare, è stato l'analisi delle funzioni invocate prima della loro definizione.

In Cinc due tipi sono equivalenti se sono lo stesso tipo o uno dei due è il tipo `star`. In particolare, quando solo uno dei due è il tipo `star`, l'unificazione di due tipi consiste nel trasformare il tipo `star` nell'altro. Ogni funzione è rappresentata internamente da una struttura dati (chiamata `Func.type`) contenente la lista dei tipi dei parametri formali ed il tipo di ritorno. Due strutture si equivalgono se sono equivalenti le rispettive liste e i rispettivi tipi di ritorno.

Tutte le volte che una funzione non ancora definita viene chiamata non è possibile effettuare controlli sulla correttezza dell'invocazione. Per non perdere le informazioni e poter posticipare il type checking, si costruisce una struttura `Func.type` che rappresenta il prototipo della funzione deducibile dalla invocazione; le informazioni disponibili riguardano unicamente i tipi dei parametri attuali mentre, per quanto riguarda il tipo di ritorno, si è scelto di impostarlo al valore generico `star`. La struttura viene memorizzata, insieme alla locazione della chiamata, in una coda specifica per quella

funzione (`IC_Queue`), in attesa che, una volta incontrata la definizione, si possa procedere ad una verifica retroattiva; per ogni invocazione errata viene mandato un messaggio di errore nel quale è specificata anche la locazione della chiamata.

Quando si incontra la definizione, dopo aver verificato che la funzione non sia già stata definita, si inserisce nella tabella dei simboli una struttura `Func_type` definitiva, in base alla quale effettuare tutti i confronti. Se l'invocazione compare all'interno di un'espressione, il tipo dell'espressione diventa un riferimento al tipo di ritorno della funzione. Non appena il tipo di questa espressione è deducibile dal contesto, esso viene unificato con quello richiesto. Mostriamo cosa succede durante il riconoscimento dell'espressione $f() + 5$ indipendentemente dal contesto in cui essa è inserita e supponendo che la funzione f non sia ancora definita. La prima regola riconosciuta è

`FunctionCall : ID ()`

In questo passaggio si crea la struttura `Func_type`. Dal contesto deduciamo che la lista dei tipi dei parametri formali contiene solo il tipo `void` mentre non abbiamo sufficienti informazioni per dedurre il tipo di ritorno che, perciò, viene impostato a `star`. Successivamente viene riconosciuta la regola

`Expression : FunctionCall`

e il tipo dell'espressione diventa un riferimento al tipo di ritorno della funzione f (cioè l'attributo `type` di `FunctionCall`). Quando, dopo qualche passaggio, si arriva alla regola

`Expression1 : Expression2 + Expression3`

si richiede (vedi Tabella 1) che le due espressioni di destra abbiano tipo intero (condizione verificata solo da `Expression3`). Pertanto, si passerà all'unificazione del tipo di `Expression2` (`star`) col tipo `int`, con conseguente modifica del tipo di ritorno della struttura `Func_type` che è presente nella coda della funzione f . Deduzioni simili a questa avvengono ogni volta che una funzione è invocata precedentemente alla sua definizione; per esempio se è invocata nella guardia di un ciclo il valore di ritorno supposto sarà `bool`, mentre se è utilizzata come indice di un array sarà `int`.

Può capitare, infine, che tra i parametri attuali di una invocazione ne compaiano alcuni di tipo `star`, per esempio nell'espressione $f(g())$ se g non è ancora definita; anche in questo caso, l'unificazione del tipo in input della funzione f comporterebbe la sostituzione del tipo di ritorno di g (`star`) per quella particolare invocazione. L'unico caso in cui non avviene nessuna deduzione sul tipo di ritorno è quando una funzione è utilizzata come comando (anche se una scelta possibile sarebbe unificare a `void`). Ovviamente il sistema di inferenza dei tipi è molto rudimentale (non esistono liste di classi di equivalenza), perché il suo scopo è solo quello di sopperire alla mancanza dei prototipi.

2.4 Error recovery

Durante le prime tre fasi (lessicale, sintattica e semantica), il compilatore segnala le situazioni di errore senza interrompere l'analisi del codice sorgente. In questo modo si possono rilevare tutti gli errori con una sola compilazione. Gli errori si possono suddividere in tre categorie:

1. **Errori lessicali:** commenti non chiusi, stringhe non chiuse o troppo lunghe, identificatori troppo lunghi, token non riconosciuti;
2. **Errori sintattici:** produzioni non riconosciute;
3. **Errori semantici:** conflitti di tipi nelle assegnazioni e nelle chiamate a funzione, utilizzo di funzioni mai dichiarate, utilizzo di identificatori non dichiarati, dichiarazioni multiple, mancanza della funzione `main`.

Per quanto riguarda gli errori sintattici, sono state aggiunte delle produzioni alla grammatica che, tramite l'utilizzo del token riservato `error`, si occupano di sollevare gli errori e scaricare automaticamente lo stack finché non risulta possibile riprendere il riconoscimento dell'input. Cinc, inoltre, comunica dei warning in corrispondenza di errori meno gravi che non pregiudicano la correttezza del programma e non impediscono la generazione del codice: variabili dichiarate e mai utilizzate, chiamate di funzione precedenti alla definizione. Per ogni messaggio viene specificata la posizione della porzione di codice errato. Quando si verificano dei conflitti di tipi (*type clash*) il messaggio di errore specifica il tipo richiesto e quello rilevato. Per esempio, consideriamo la definizione di funzione

```
func Foo: int a, bool b -> void : .
```

e l'invocazione errata `Foo(1,2)`. Cinc genererà il seguente messaggio:

```
Func.cinc[3] Function Mismatch Error
                (line 4, col 5 - line 4, col 13)
Invocation of function 'Foo' does not respect definition
                (line 1, col 1 - line 1, col 31)
required: int * bool -> void
found: int * int -> void
```

Le prime coordinate indicano la locazione dell'invocazione mentre le seconde permettono di risalire alla definizione di `Foo`.

2.5 Generazione del codice

L'ultima fase del front-end riguarda la generazione del codice intermedio; esso verrà utilizzato dal back-end per generare codice oggetto. Il vantaggio di generare codice intermedio consiste nel poter utilizzare back-end differenti, ognuno con diverse caratteristiche (in riguardo all'ottimizzazione o all'architettura hardware di destinazione).

La scelta del back-end è ricaduta su Jasmin, un assembler per Java. L'input di Jasmin è un semplice file di testo in cui è contenuta la descrizione delle classi Java per mezzo del set di istruzioni della Java Virtual Machine. L'output è un file binario, contenente il byte-code di una classe Java, che può essere eseguito per mezzo di un interprete. Una peculiarità di Jasmin riguarda la possibilità di istanziare qualsiasi oggetto definito nelle API Java e di utilizzarne tutti i metodi disponibili, aprendo la strada a possibili evoluzioni future. Questo ha permesso di gestire in modo nativo le stringhe (e di conseguenza le operazioni su di esse, come la concatenazione), i vettori multidimensionali ed il comando di output a video (`print`).

L'assembler Java si basa su una macchina a pila: la maggior parte delle istruzioni richiedono di posizionare precedentemente sulla cima di uno stack i valori o i riferimenti necessari. Per esempio: prima di eseguire una somma è necessario inserire il codice che mette sullo stack entrambi gli operandi e poi quello per richiamare il comando di somma; a run-time lo stack conterrà prima i due valori da sommare e successivamente il loro risultato, che dovrà poi essere gestito nel modo opportuno, poiché non risulta ancora salvato in nessun registro.

I tipi delle variabili presenti in Cinc sono gestiti nel modo seguente: gli interi e i booleani sono trattati come valori primitivi interi, le stringhe sono degli oggetti il cui spazio in memoria viene allocato automaticamente a run-time, mentre per i vettori è necessario inserire appositi comandi assembler per allocarne la memoria necessaria (la loro dimensione può essere calcolata a run-time e posizionata sullo stack). I valori (primitivi o riferimenti ad oggetti) possono essere memorizzati in appositi registri.

Per generare il file di assembler è necessario visitare l'albero sintattico astratto. Il primo passo di questa fase è quello di emettere il codice che si occupa di creare una classe; ogni programma scritto in Cinc, infatti, dà origine ad un unico file contenente una classe il cui nome è ricavato da quello del file togliendo l'estensione '.cinc'. Il file di codice intermedio avrà estensione '.j'.

Ogni funzione di un programma in Cinc viene tradotta come un metodo della classe; le variabili globali invece diventano variabili di istanza della classe e sono quindi visibili a tutti i suoi metodi. Jasmin permette di richiamare le variabili globali semplicemente tramite il loro nome che, per coerenza e semplicità, è lo stesso usato dal programmatore nel codice sorgente. Le variabili locali alle funzioni sono invece memorizzate nei registri il cui campo di visibilità è limitato ad ogni singolo metodo; il loro identificatore è semplicemente un numero intero, per cui è stato necessario utilizzare una tabella dei simboli analoga a quella usata nelle fasi precedenti del front-end. Essa gestisce in modo opportuno gli ambienti annidati e permette di associare ad ogni variabile l'identificatore numerico del registro in cui è contenuto il suo valore. La prima volta che viene trovata una variabile, vengono inseriti gli opportuni valori nella tabella dei simboli in modo da ritrovarli velocemente

ad ogni utilizzo successivo (evitando di dover rivisitare l'albero). Inoltre, nel caso dei vettori, è utile anche per memorizzarne le dimensioni ed il tipo semplice che vi è contenuto.

2.5.1 Il numero di registri

Il numero di registri necessari, così come la grandezza dello stack, è diverso per ogni funzione e può essere calcolato staticamente durante la visita dell'albero. Jasmin mette a disposizione due direttive (`.limit locals` e `.limit stack` seguite da un argomento con valore intero maggiore o uguale a zero) la cui funzione è proprio quella di specificare staticamente la quantità di spazio che la Java Virtual Machine dovrà assegnare a run-time per ogni invocazione di un metodo. Il calcolo deve essere eseguito con precisione per ottimizzare il più possibile il byte-code prodotto dal back-end.

Per quanto riguarda il numero dei registri necessari è stata creata un'apposita funzione (`AA_GetLocals`) che viene applicata ad ogni nodo dell'albero astratto che rappresenta il corpo di una funzione. `AA_GetLocals` itera sulla lista dei comandi presenti, cercando tutte le dichiarazioni e richiamandosi ricorsivamente se sono presenti blocchi annidati (per esempio il corpo di un ciclo `while`). La peculiarità consiste nel non incrementare banalmente un contatore ogni volta che trova una dichiarazione, ma nel valutare il numero massimo di variabili che possono essere istanziate contemporaneamente. Per esempio:

```
func foo: void -> void :
    int a;
    :
        int b,c;
    .
    :
        int d,e;
    .
.
```

La funzione `foo` richiede solamente lo spazio per 3 registri e non 5 come si potrebbe erroneamente pensare, perché le variabili `b` e `c` non servono più nel momento in cui si esce dal blocco che le ha dichiarate. Da notare che il numero di registri sarebbe 3, per lo stesso identico motivo, anche se `d` ed `e` non fossero state dichiarate dentro un blocco annidato.

È simpatico notare come la funzione `AA_GetLocals`, in verità, proceda a ritroso, analizzando il codice sorgente nell'ordine inverso rispetto a come è stato scritto. Nell'esempio precedente analizzerebbe prima l'ultimo blocco annidato (quello con le dichiarazioni di `d` ed `e`), poi l'altro blocco ed infine la dichiarazione di `a`. Questa particolarità permette però di semplificare

molto la funzione, calcolando ogni volta il massimo fra lo spazio necessario fino a quel momento e quello necessario al blocco che incontra risalendo il codice. Al numero di registri ottenuto in questo modo bisogna sommare il numero degli eventuali parametri formali che ha la funzione (in quanto sono dichiarazioni a tutti gli effetti, con visibilità per l'intero corpo della funzione). Inoltre viene aggiunto spazio per due ulteriori registri che vengono utilizzati per semplificare il codice assembler generato in presenza di particolari comandi o operatori (come l'elevamento a potenza).

2.5.2 La dimensione dello stack

Valutare la dimensione ottimale dello stack è leggermente più complesso: ha richiesto lo sviluppo di numerose funzioni, sia iterative che ricorsive, in grado di calcolare la quantità di memoria necessaria per eseguire ogni comando e risolvere ogni espressione. Questa quantità dipende non solo dal codice presente nel file sorgente, ma anche da come esso viene tradotto in assembler Java. Per esempio, l'operazione di elevamento a potenza è stata implementata limitando l'uso dello stack, ma utilizzando due registri aggiuntivi in cui vengono mantenuti i risultati parziali delle moltiplicazioni; un'implementazione alternativa avrebbe potuto fare uso del solo stack, ma richiedendone probabilmente una dimensione maggiore (ed anche un maggior numero di istruzioni). Inoltre bisogna sempre stare attenti, per ogni comando presente nel file sorgente, a generare codice assembler che non lasci mai nulla sulla pila dopo la sua esecuzione. Per esempio, il comando relativo al ciclo `for` durante tutta la sua esecuzione mantiene sullo stack un valore intero (utilizzato per valutare la necessità di eseguire un ulteriore ciclo). Un eventuale errore potrebbe diventare noto solamente in condizioni particolari, per esempio all'interno di cicli `while` o `for` che contengono un comando errato, perché l'errore verrebbe amplificato e potrebbe sollevare eccezioni di *stack overflow* a run-time. Questi problemi hanno quindi richiesto un cura particolare, soprattutto nella fase di debugging del compilatore.

2.5.3 Backpatching

Dopo aver valutato il numero di registri e la dimensione dello stack si procede con l'emissione del codice relativo a tutte le istruzioni presenti nel corpo della funzione che si sta analizzando.

La generazione del codice utilizza la tecnica del *backpatching*; ogni espressione booleana mantiene due liste che contengono l'elenco delle etichette (ovvero gli argomenti dei `goto`) alle quali bisogna saltare nel caso in cui il risultato sia vero o falso. Quando non si è in grado di emettere il codice relativo al salto ad una particolare etichetta, perché non è ancora stata definita, si genera un'etichetta fittizia (utile solo per riservare lo spazio necessario nel file). Le posizioni nel file in cui bisognerà inserire le etichette corrette

vengono memorizzate nelle liste delle espressioni booleane. In questo modo si genera il codice in una sola passata perché è sufficiente aggiornare solo le etichette tralasciate in precedenza. Le etichette sono formate dal nome della funzione, da un numero intero monotono crescente e da alcuni caratteri speciali in modo da ottenere identificatori sempre della stessa lunghezza (almeno all'interno di ogni funzione), e semplificarne la gestione in fase del backpatching. Per esempio, alcuni identificatori nel codice della funzione `foo` potrebbero essere: `@_foo__1000_@`, `@_foo__1001_@`, `@_foo__1002_@`, etc. . .

Una nota particolare va alla traduzione del comando `print` di Cinc; il compilatore, quando incontra un'espressione booleana, genera del codice assembler, utilizzando la tecnica del backpatching, per far sì che a run-time vengano emesse in output le stringhe `true` e `false` piuttosto che i valori interi 1 e 0 gestiti da Jasmin.

A La grammatica

```
Program : FunctionList
        | Global FunctionList
        ;

Global : Declaration ";"
       | Global Declaration ";"
       ;

Declaration : Type IdList
            ;

Type : SimpleType
     | SimpleType ArrayDim
     ;

SimpleType : "bool"
           | "int"
           | "string"
           ;

ArrayDim : "[" "]"
         | ArrayDim "[" "]"
         ;

IdList : ID
       | IdList "," ID
       | IdList ID
       ;

FunctionList :
           | FunctionList Function
           ;

Function : FunctionDef Block
        ;

FunctionDef : "func" ID ":" BlockIn FormalParam
            "->" ReturnTyp
            ;

FormalParam : "void"
            | ParamList
```

```

;

ParamList : Type ID
          | ParamList "," Type ID
          ;

ReturnType : "void"
           | Type
           ;

BlockIn :
        ;

Block : ":" StatementList "."
      ;

StatementList :
            | StatementList Statement
            ;

Statement : BlockIn Block
          | Declaration ";"
          | FunctionCall ";"
          | AssignStatement
          | PrintStatement
          | ReturnStatement
          | IfThenStatement
          | IfThenElseStatement
          | WhileStatement
          | ForStatement
          ;
```

```
AssignStatement : ID "=" Expression ";"
                | ID ArrayIndex "=" Expression ";"
                ;

ArrayIndex : "[" Expression "]"
           | ArrayIndex "[" Expression "]"
           ;

FunctionCall : ID "(" ")"
             | ID "(" ExpressionList ")"
             ;

PrintStatement : "print" ExpressionList ";"
               ;

ReturnStatement : "return" Expression ";"
                ;

IfThenStatement : "if" Expression BlockIn Block
                ;

IfThenElseStatement : "if" Expression BlockIn Block
                    "else" BlockIn Block
                    ;

WhileStatement : "while" Expression BlockIn Block
               ;

ForStatement : "for" BlockIn ID Expression "->" Expression
              Block
              ;
```

```
ExpressionList : Expression
                | ExpressionList "," Expression
                ;
```

```
Expression : Expression "+" Expression
            | Expression "-" Expression
            | Expression "*" Expression
            | Expression "/" Expression
            | Expression "mod" Expression
            | Expression "exp" Expression
            | Expression "\^{}" Expression
            | "abs" Expression
            | "-" Expression
            | Expression "and" Expression
            | Expression "or" Expression
            | "not" Expression
            | Expression "<" Expression
            | Expression "<=" Expression
            | Expression ">" Expression
            | Expression ">=" Expression
            | Expression "==" Expression
            | Expression "!=" Expression
            | "(" Expression ")"
            | FunctionCall
            | ID ArrayIndex
            | "new" SimpleType ArrayIndex
            | ID
            | CONST\_BOOL
            | CONST\_INT
            | CONST\_STRING
            ;
```

Indice

Prefazione	1
1 Il Linguaggio	2
1.1 Principi fondamentali	2
1.2 Dichiarazioni	2
1.3 Funzioni	3
1.4 Espressioni	3
2 L'implementazione	5
2.1 Analisi lessicale	5
2.2 Analisi sintattica e tabella dei simboli	5
2.3 Analisi semantica e deduzione dei tipi	6
2.4 Error recovery	8
2.5 Generazione del codice	8
2.5.1 Il numero di registri	10
2.5.2 La dimensione dello stack	11
2.5.3 Backpatching	11
A La grammatica	13