Elaborato di linguaggi e traduttori

Gruppo: Granata-Merenda

Progetto

Testo

Costruire un compilatore per il linguaggio MiniC) che generi istruzioni per la Java Virtual Machine.

Per la creazione di scanner e parser possono scegliere il linguaggio di programmazione che si preferisce (java, c, c#, c++). Possono essere utilizzati esclusivamente I tool Lex e Yacc o JLex e JCup per la generazione di scanner e parser.
Il compilatore deve emettere un messaggio di errore appropriato nell'eventuali che rilevi la presenza di errori lessicali e sintattici.
L'analisi lessicale e sintattica non deve arrestarsi al primo errore rilevato ma deve consentire la segnalazione di tutti gli errori presenti.

Il compilatore dovrà accettare un file contenente un programma scritto nel linguaggio miniC e dovrà scrivere il risultato della compilazione in un file che ha lo stesso nome del file sorgente, con l'estensione cambiata da .c a .j. Per esempio, supponendo che l'eseguibile del compilatore abbia nome **minic**, il comando minic file.c deve produrre un file file.j contente codice Jasmin.

Successivamente, il file .j potrebbe essere processato con Jasmin per ottenere il .class: jasmin file.j
A questo punto è possibile eseguire il programma compilato invocando l'interprete Java.

IL linguaggio Mini C

Il linguaggio MiniC è una variante semplificata e "ripulita" del linguaggio C . La sintassi del MiniC è fornita di seguito.

```
<gdecl> ::= <fun>
<fun> ::= extern string <type> id ( <type> {,<type>} ) ;
        | <type> id ( <tid> [, <tid>] ) <block>
<tid>   ::= <type> id
<ldec> ::= <tid > ;
<const> ::= bool | int | float | string
<type> ::= void | bool | int | float | string | <type>[]
<expr> ::= <const> | <aexpr> | id ( <expr> {,<expr> ) | new <type> [ <expr> ]
        | ( <type> ) <expr> | <unop> <expr> | <expr>1 <binop> <expr> | ( <expr> )
<aexpr> ::= id | <aexpr> [ <expr> ]
<stmt> ::= <expr> ; | <aexpr> = <expr> ; | if ( <expr> ) <stmt> [else <stmt>]
        | while ( <expr> ) <stmt> | return [<expr>] ; | <block>
<block> ::= { <ldecl>  {<ldecl>} <stmt> {<stmt> }
<unop> ::= ! | + | -
<binop< ::= + | - | * | / | % | && | || | == | != | <= | < | > | >=
```

La sintassi delle costanti e dei commenti deve essere la stessa del linguaggio C;

Il linguaggio miniC include i seguenti tipi primitivi:

- bool : il tipo dei valori booleani true e false;
- int: il tipo dei numeri interi;
- float: il tipo dei numeri in virgola mobile a precisione singola;
- string: il tipo delle stringhe di caratteri.

In aggiunta, è presente il tipo strutturato T[] che definisce il tipo degli array (di lunghezza non specificata) i cui elementi sono di tipo T. Notare che `e possibile definire array di array, ecc. Il tipo void non è un vero e proprio tipo di dato, nel senso che non esiste alcun valore che ha tipo void. Esso viene usato esclusivamente per denotare il tipo di funzioni che non ritornano risultati.

La verifica statica dei tipi di espressioni e comandi è parametrizzata da un contesto. Useremo la lettera Δ per indicare un contesto, cioè un insieme di associazioni V : T dove V è il nome di una variabile e T e il tipo ad essa associata ed indicheremo con $\Delta(V)$ il tipo associato al simbolo V nel contesto Δ. Il contesto iniziale Δ_0 contiene le dichiarazioni di tipo degli operatori predefiniti del linguaggio miniC nonchè i tipi delle costanti. Le varianti unarie degli operatori + e - sono indicate con +u e –u rispettivamente.

S	$\Delta_0(S)$
false, true	Bool
n	int
f	float
s	string
!	bool → bool
+u, -u	{int, float} → {int, float}
+, -, *, /, %	({int, float}, {int, float}) → {int, float}
&&, ||	(bool , bool) → bool
==, !=	({bool , int, float}, {bool , int, float}) → bool
<, <=, >, >=	({int, float}, {int, float}) → bool

Tabella 1

La tabella 2 mostra le regole di type checking per le espressioni di miniC. Ogni regola è da leggersi dall'alto verso il basso: sopra la riga compaiono le premesse, sotto la riga la conclusione. Per esempio, la regola

[UNOP] $\Delta \vdash$ <unop> : $T_1 \rightarrow T_2$ $\Delta \vdash E : T_1$
$\Delta \vdash$ <unop> E : T_2

stabilisce che se nel contesto Δ l'espressione E ha tipo T_1 e l'operatore unario <unop> ha tipo $T_1 \rightarrow T_2$ (in base alla tabella 2) allora l'espressione <unop> E_2 ha tipo T_2.

[CONST]	$\Delta \vdash k : \Delta(k)$
[VAR]	$\Delta \vdash V : \Delta(V)$
[PROM]	$\Delta \vdash E$: int $\Delta \vdash E$: float
[ARRAY]	$\Delta \vdash V : T[]$ $\Delta \vdash E$: int $\Delta \vdash V[E] : T$
[CALL]	$\Delta \vdash V : (T1, \ldots Tn) \rightarrow T$ $\Delta \vdash E1 : T1 \ldots \Delta \vdash En : Tn$ $\Delta \vdash EV(E1 .. En) : T$
[NEW]	$\Delta \vdash E$: int $\Delta \vdash$ new T [E] : T[]

[CAST]	Δ |– E : float Δ |– (int) E : int
[UNOP]	Δ |– <unop> : T1 → T2 Δ |– E : T1 Δ |– <unop>E : T2
[BINOP]	Δ |– <binop> : (T1,T2) → T Δ |– E1 : T1 Δ |– E2 : T2 Δ |– E1 <binop> E2 : T

Tabella 2. Regole di type checking

Si noti come la regola [PROM] consenta di "promuovere" un valore di tipo intero ad uno di tipo floating point a precisione singola. Tale regola non va applicata sempre e indiscriminatamente, altrimenti una espressione come 1 + 2 verrebbe sempre compilata eseguendo una somma tra valori floating point che, oltre ad essere più dispendiosa in termini computazionali, può anche risultare meno accurata della somma tra numeri interi a causa di errori di approssimazione. Si intende che il compilatore deve promuovere un valore da intero a floating point solo quando `e strettamente necessario. Si noti infine che la conversione inversa da floating point a intero non è mai eseguita implicitamente dal compilatore, ma deve essere il programmatore a richiederla attraverso l'uso dell'operatore di cast.

La tabella 3 mostra le regole di type checking per i comandi di miniC. Le regole si leggono come quelle per il type checking delle espressioni.
Tuttavia, non restituendo alcun valore i comandi "non hanno tipo". Per cui, in ogni regola indichiamo con S : □ il fatto che un comando S sia ben formato, cio`e che rispetti un insieme di vincoli relativi parti che lo compongono. Per esempio, la regola

[WHILE] Δ |– E : bool Δ |– S : □
Δ |– while (E) : S : □

stabilisce che un costrutto while (E) S è ben formato se E è un'espressione di tipo bool e se S è a sua volta ben formato.

[IGNORE]	Δ |– E : void Δ |– E; : □
[ASSIGN]	Δ |– A : T Δ |– E : T Δ |– A =E; : □
[IF]	Δ |– E : bool Δ |– S1 : □ [Δ |– S1 : □] Δ |– if (E) S1 [else S2] : □
[RETURN]	[Δ |– E : T Δ |– return [E] ; : □
[WHILE]	Δ |– E : bool Δ |– S : □ Δ |– while (E) S : □
[BLOCK]	Δ [V1 : T1] + · · · + [Vn : Tn] |– Si : □ per ogni 1<=i<=m Δ |– { T1 V1; · · · Tn Vn; S1 · · · Sm} : □

Tabella 3

Infine, la regola di type checking per le funzioni è la seguente

$$\frac{\Delta + [V_1 : T_1] + \cdots + [V_n : T_n] + [F : (T_1, \ldots, T_n) \rightarrow T] \vdash S : \square}{\Delta \vdash T\ F(T_1\ V_1, \ldots, T_n\ V_n)\ S \quad \text{è una funzione ben formata}}$$

L'analisi statica che il compilatore deve implementare deve includere tutti i controlli di tipo descritti in questa sezione, nonch´e la verifica che tutti gli identificatori utilizzati siano stati effettivamente dichiarati e siano visibili secondo le regole di scoping del linguaggio miniC, che sono del tutto analoghe a quelle del linguaggio C.

Ulteriori controlli di semantica statica sono

1. ogni variabile locale venga inizializzata prima del suo utilizzo. In caso contrario, generare un messaggio di avvertimento.
2. verificare il corretto ritorno di un valore per una funzione.

La macchina astratta

La macchina astratta da usare per la compilazione di programmi in miniC è la Java Virtual Machine. Le istruzioni che possono essere usate sono le seguenti:

Caricamento di costanti			
LoadInt(n)		n : Int	carica la costante intera n
LoadFloat(f)		f : Float	carica la costante in virgola mobile a precisione singola f
LoadString(s)		s : String	carica la costante stringa s
Accesso alle variabili locali			
Load(t, i)		v : t	carica il valore di tipo t della i-esima variabile locale
Store(t, i)		v : t	memorizza il valore v di tipo t nella i-esima variabile locale
Conversioni di tipo			
Convert(t1, t2)	v1 : t1	v2 : t2	converte il valore v1 di tipo t1 nel valore v2 di tipo t2. Le conversioni consentite sono Int ! Float Int ! Double,Float ! Double,Float ! Int,Double ! Int,Double ! Float
Operazioni aritmetiche			
Neg(t)	v1 : t v2 :t	v : t	calcola la negazione −v1
Add(t)	v1 : t v2 :t	v : t	calcola la somma v1 + v2
Sub(t)	v1 : t v2 :t	v : t	calcola la differenza v1 - v2
Mul(t)	v1 : t v2 :t	v : t	calcola la moltiplicazione v1 * v2
Div(t)	v1 : t v2 :t	v : t	calcola la divisione v1 : v2
Rem(t)	v1 : t v2 :t	v : t	calcola il resto della divisione v1 : v2
Operazioni logiche			
And	v1 : int v2 :int	v : int	calcola la congiunzione logica v1 and v2
Or	v1 : int v2 :int	v : int	calcola la disgiunzione logica v1 OR v2
Operazioni su array			
NewArray(t)	n : int	a : Array(t)	crea un nuovo array la cui dimensione è n e il tipo degli elementi è t
ArrayLoad(t)	a : Array(t)	n : Int v : t	carica l'n-esima componente dell'array a
ArrayStore(t)	a : Array(t)	n : Int v : t	memorizza il valore v nell'n-esima componente dell'array a
Istruzioni di salto			
JumpIf(c, t, l)	v1 : t v2 : t		confronta i due valori v1 e v2 di tipo t e salta ad l se i due valori soddisfano la relazione c 2 {Eq, Ne, Lt, Le, Gt, Ge}

Jump(l)			salta a l in modo incondizionat
Funzioni			
Call(p, [t1; · · · ; tn], t)	v1 : t1 …vn : tn	v : t	chiama la funzione con nome p passando parametri v1 · · · vn. v `e il valore ritornato dalla funzione, ed `e presente solo se t = void
Return(t)	v : t		termina la funzione corrente ritornando il valore v di tipo t. Se t = Void non viene ritornato alcun valore (e lo stack deve essere vuoto)
Gestione dello stack			
Pop(t)	v : t		rimuove il valore in cima allo stack

Modalita di consegna

L'elaborato va consegnato sul sito del corso almeno 4 giorni prima della data effettiva del colloquio orale.

Vanno consegnati:

- Codice comprese eventuali librerie utilizzate
- eventuali file di prova
- documentazione che descrive le scelte implementative
- README

Nel caso di gruppi composti da più di una persona va indicato in maniera chiara se il progetto è stato suddiviso fra i diversi componenti. Il colloquio orale può essere sostento in date diversa da parte dei diversi componenti del gruppo

Eventuali Link

La documentazione completa si trova al l'indirizzo
http://jasmin.sourceforge.net/