

5. Linguaggi Formali e Traduttori

5.4 Traduzione di espressioni aritmetiche

- Sommario
- Grammatica delle espressioni aritmetiche
- SDD per espressioni aritmetiche
- SDT per la grammatica ambigua
- SDT “on-the-fly” per la grammatica LL(1)
- Esercizi

È proibito condividere e divulgare in qualsiasi forma i materiali didattici caricati sulla piattaforma e le lezioni svolte in videoconferenza: ogni azione che viola questa norma sarà denunciata agli organi di Ateneo e perseguita a termini di legge.

Sommario

Problema

- Traduzione delle espressioni aritmetiche.

In questa lezione

- Definiamo SDD e SDT per la traduzione di espressioni aritmetiche.

Riferimenti esterni

- [Java Language and Virtual Machine Specifications](#)
- [JVM Instruction set](#)

Grammatica delle espressioni aritmetiche

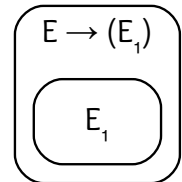
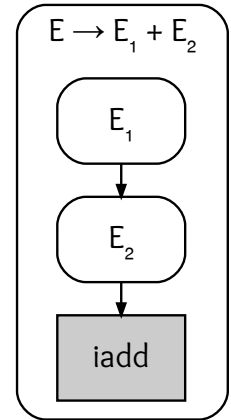
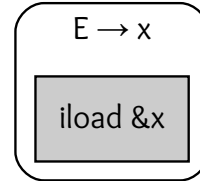
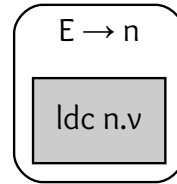
Produzioni	Descrizione
$E \rightarrow E_1 + E_2$	Somma
$E \rightarrow E_1 - E_2$	Sottrazione
$E \rightarrow E_1 * E_2$	Moltiplicazione
$E \rightarrow E_1 / E_2$	Divisione intera
$E \rightarrow E_1 \% E_2$	Resto della divisione intera
$E \rightarrow (E_1)$	Stesso valore di E_1
$E \rightarrow \mathbf{n}$	Costante
$E \rightarrow \mathbf{x}$	Variabile

Attenzione

Al fine di presentare la traduzione in codice intermedio adotteremo spesso grammatiche ambigue o comunque non LL(1). Disambiguazione, fattorizzazione, eliminazione della ricorsione spesso richiedono modifiche significative anche agli SDT corrispondenti, come l'introduzione di attributi ereditati.

SDD per espressioni aritmetiche

Produzioni	Regole semantiche
$E \rightarrow n$	$E.code = ldc\ n.v$
$E \rightarrow x$	$E.code = iload\ \&x$
$E \rightarrow E_1 + E_2$	$E.code = E_1.code$ $\parallel E_2.code$ $\parallel iadd$
$E \rightarrow (E_1)$	$E.code = E_1.code$

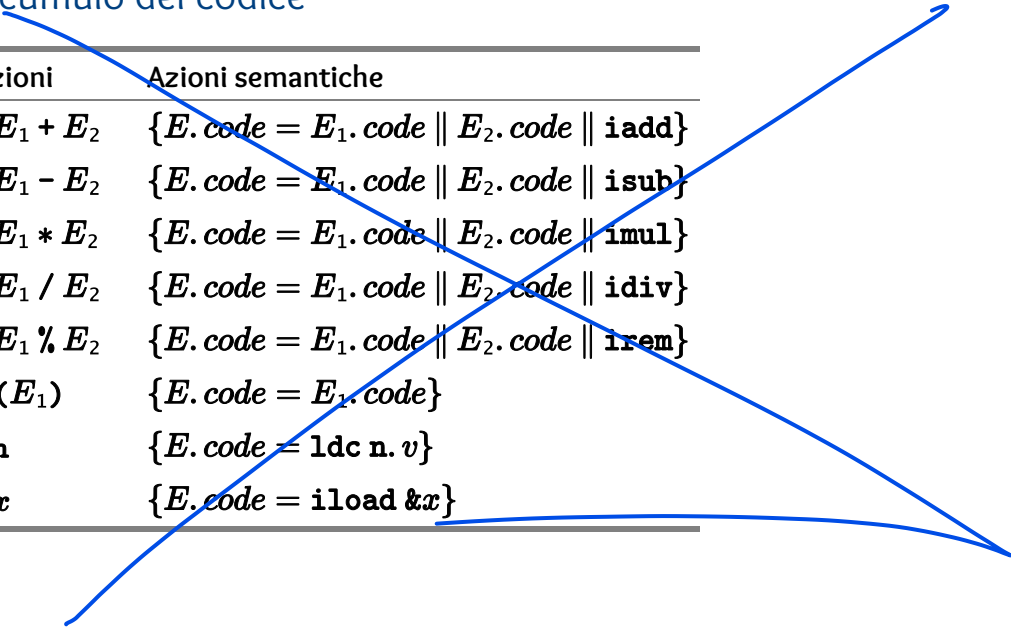


Attributi

- $E.code$ = codice che calcola il valore di E e lo lascia in cima alla pila.

SDT per la grammatica ambigua

SDT con accumulo del codice



Produzioni	Azioni semantiche
$E \rightarrow E_1 + E_2$	$\{E.code = E_1.code \parallel E_2.code \parallel \text{iadd}\}$
$E \rightarrow E_1 - E_2$	$\{E.code = E_1.code \parallel E_2.code \parallel \text{isub}\}$
$E \rightarrow E_1 * E_2$	$\{E.code = E_1.code \parallel E_2.code \parallel \text{imul}\}$
$E \rightarrow E_1 / E_2$	$\{E.code = E_1.code \parallel E_2.code \parallel \text{idiv}\}$
$E \rightarrow E_1 \% E_2$	$\{E.code = E_1.code \parallel E_2.code \parallel \text{irem}\}$
$E \rightarrow (E_1)$	$\{E.code = E_1.code\}$
$E \rightarrow n$	$\{E.code = \text{ldc } n.v\}$
$E \rightarrow x$	$\{E.code = \text{iload } \&x\}$

SDT per la grammatica ambigua

SDT con accumulo del codice

Produzioni	Azioni semantiche
$E \rightarrow E_1 + E_2$	$\{E.code = E_1.code \parallel E_2.code \parallel \text{iadd}\}$
$E \rightarrow E_1 - E_2$	$\{E.code = E_1.code \parallel E_2.code \parallel \text{isub}\}$
$E \rightarrow E_1 * E_2$	$\{E.code = E_1.code \parallel E_2.code \parallel \text{imul}\}$
$E \rightarrow E_1 / E_2$	$\{E.code = E_1.code \parallel E_2.code \parallel \text{idiv}\}$
$E \rightarrow E_1 \% E_2$	$\{E.code = E_1.code \parallel E_2.code \parallel \text{irem}\}$
$E \rightarrow (E_1)$	$\{E.code = E_1.code\}$
$E \rightarrow \mathbf{n}$	$\{E.code = \text{ldc n.v}\}$
$E \rightarrow x$	$\{E.code = \text{iload \&x}\}$

SDT “on-the-fly”

Azioni semantiche
$\{\text{emit}(\text{iadd})\}$
$\{\text{emit}(\text{isub})\}$
$\{\text{emit}(\text{imul})\}$
$\{\text{emit}(\text{idiv})\}$
$\{\text{emit}(\text{irem})\}$
$\{\text{emit}(\text{ldc n.v})\}$
$\{\text{emit}(\text{iload \&x})\}$

SDT “on-the-fly” per la grammatica LL(1)

Produzioni	Azioni semantiche
$E \rightarrow TE'$	
$E' \rightarrow \varepsilon$	
$E' \rightarrow +T$ E'	$\{emit(iadd)\}$
$E' \rightarrow -T$ E'	$\{emit(isub)\}$
$T \rightarrow FT'$	
$T' \rightarrow \varepsilon$	
$T' \rightarrow *F$ T'	$\{emit(imul)\}$
...	
$F \rightarrow n$	$\{emit(ldc\ n.v)\}$
$F \rightarrow x$	$\{emit(iload\ \&x)\}$
$F \rightarrow (E)$	

```
private void E'() {
    switch (peek()) {
        case '+': // E' → +TE'
            match('+');
            T();
            emit("iadd");
            E'();
            break;
        case '-': // E' → -TE'
            match('-');
            T();
            emit("isub");
            E'();
            break;
        case ')':
        case '$': // E' → ε
            break;
        default:
            throw error("E'");
    }
}
```

Esercizi

1. Calcolare il codice generato per l'espressione $x * x + 2 * x + 1$.
2. Scrivere le regole semantiche per tradurre la negazione $E \rightarrow -E_1$.
3. Scrivere le regole semantiche per tradurre l'accesso ad array $E \rightarrow E_1[E_2]$.
4. Scrivere le regole semantiche per tradurre l'invocazione di un metodo statico

$$E \rightarrow m(E_{list}) \quad E_{list} \rightarrow \varepsilon \mid E_{listp} \quad E_{listp} \rightarrow E \mid E, E_{listp}$$

usando l'istruzione **invokestatic** m per invocare m .

5. Scrivere le regole semantiche per tradurre l'assegnamento $E \rightarrow x = E_1$ ricordando che, in Java, tale comando è anche un'espressione il cui valore coincide con quello di E_1 . **Suggerimento:** usare l'istruzione `dup` per evitare di valutare E_1 due volte.
6. Scrivere le regole semantiche per tradurre pre- e post-incremento di variabili

$$E \rightarrow ++x \mid x++$$

tenendo presente la differente semantica delle due forme.

7. Scrivere le regole semantiche per la traduzione del post-incremento dell'elemento di un array $E \rightarrow E_1[E_2]++$ ricordando che il valore di tale espressione è quello dell'elemento prima dell'incremento. **DIFFICILE!** Per risolvere l'esercizio usare `dup2` e `dup_x2`.