Compilatori

Corso di Laurea in Informatica

Mauro Leoncini

A.A. 2024/2025

1. Il Linguaggio Funzionale Minimo (LFM)
   - Caratteristiche generali
   - La grammatica di LFM
   - Abstract Syntax Tree per LFM

Compilatori

Caratteristiche del linguaggio

- L'unico tipo di dato del linguaggio sono i numeri interi
- I due unici meccanismi di controllo sono funzioni ricorsive e costrutto condizionale (a più vie)
- Non c'è visibilità dello store e dunque non esistono variabili né assegnamento
- Esiste però un costrutto `let`, presente in quasi tutti i linguaggi funzionali, che consente di definire un blocco preceduto da uno o più "assegnamenti" di valori ad identificatori
- Il valore di un identificatore non può però essere cambiato all'interno del blocco `let` (e non è più referenziabile al di fuori del blocco stesso)
- Per le funzionalità di I/O è disponibile la definizione di elementi esterni

Primi esempi

- Come primo esempio, presentiamo una funzione ricorsiva per il calcolo del fattoriale

```
function fact(n)
    if n==0 : 1;
       true : n*fact(n-1)
    end
end
```

- Il secondo esempio mostra l'uso tipico del costrutto `let`

```
function fibo(n)
    if n<=1 : n;
       true : let x=fibo(n-2),
                  y=fibo(n-1) in
                  x+y
              end
    end
end
```

Un "programma" completo

- Programma completo per il calcolo dell'n-esimo numero di Fibonacci
- Il numero n viene letto e il valore `fibo`(n) stampato mediante chiamata a funzioni esterne (scritte in C)

```
external readint()
external printint(n)
function fibo(n)
    if n<=1 : n;
       true : let x=fibo(n-2),
                  y=fibo(n-1) in
                  x+y
              end
    end
end
let n=readint(),
    F=fibo(n) in
    printint(F)
end
```

Compilatori

Grammatica completa

- La grammatica completa, inclusa la specifica dei token, si trova nel file `lfm.lr`
- Nel seguito discutiamo brevemente come viene fatto il trattamento delle epressioni aritmetiche e di quelle logiche

Grammatica

- Come già detto, la parte della grammatica per le espressioni aritmetiche utilizza le produzioni che in teoria sono ambigue
- La ragione (anche questa già sottolineata) è che così facendo si riduce sensibilmente il numero di riduzioni
- Bison utilizza le direttive `%left` e `%right` per stabilire se un operatore è associativo a sinistra (come le 5 operazioni aritmetiche) o a destra (come l'esponenziazione)
- L'ordine di inclusione nel file bison di tali direttive determina poi il livello di precedenza, in ordine crescente
- Per riflettere correttamente le precedenze assunte in Matematica bisogna quindi includere le seguenti direttive nell'ordine indicato

```
%left "+" "-";
%left "*" "/" "%";
```

Trattamento delle espressioni

- La direttive consentono al perser di risolvere i conflitti shift-reduce che si verificano nel caso più operatori compaiano nella stessa espressione
- Ad esempio, in fase di riconoscimento dell'espressione $\texttt{num} + \texttt{num} * \texttt{num}$, il parser arriva alla forma di frase $E + E * \texttt{num}$ dove $E + E$ è sullo stack e $*\texttt{num}$ ancora in input
- A questo punto si verifica un conflitto perché potrebbe essere corretto tanto effettuare la riduzione (considerando $E + E$ come handle) quano uno shift del simbolo $*$
- La direttiva, che attribuisce maggiore precedenza all'operatore $*$ rispetto al $+$, informa il parser che deve effettuare lo shift
- Se invece l'espressione fosse $\texttt{num} + \texttt{num} + \texttt{num}$, arrivato allo stesso punto il parser effettuerebbe la riduzione perché la direttiva che riguarda il token $+$ specifica che esso è "left-associative"

Trattamento delle espressioni

- Esiste però ancora il problema degli *operatori unari* e, segnatamente, del meno unario
- In Matematica come deve essere valutata l'espressione $-3-2$? Il suo valore deve essere $-5 = (-3) - 2$, oppure $-1 = -(3-2)$?
- Il valore corretto (secondo le convenzioni universalmente utilizzate in Matematica) è che il valore debba esser -5, perché si attribuisce al meno unario una precedenza più elevata rispetto agli altri operatori
- Il problema è che il token utilizzato è lo stesso del meno "binario" e questo ha precedenza più bassa rispetto a moltiplicazione e divisione.
- In altri termini, la direttiva che riguarda il token $-$ dovrebbe stare per un verso prima e per un verso dopo la direttiva che riguarda il $*$
- C'è poi da dire che il meno unario non è un operatore associativo

Trattamento delle espressioni

- È necessario osservare che Bison permette di attribuire una priorità non solo ai token (in particolare agli operatori) ma anche alle produzioni
- Questo permette di risolvere in modo diretto i conflitti shift-reduce perché il parser può confrontare la priorità della produzione e quella del token da (eventualmente) "shiftare"
- Per attribuire una prioritá ad una produzione si usa (nella produzione stessa) la direttiva `%prec`
- La soluzione consiste quindi nel definire un livello di precedenza per un token fittizio e poi indicare quel token nella direttiva `%prec`
- Ad esempio, possiamo innanzitutto scrivere

```
...
%left "+" "-";
%left "*" "/" "%";
%nonassoc UMINUS;
```

Trattamento delle espressioni

- In questo modo abbiamo stabilito che il token `UMINUS` (che non vedremo mai nell'input) non è associativo ma ha una priorità superiore a quella di $*$ e $/$
- Siamo quindi in grado di attribuire quella priorità ad una regola facendo riferimento proprio a `UMINUS`

```
...
%left "+" "-";
%left "*" "/" "%";
%nonassoc UMINUS;
...
exp:
...
| "-" exp %prec UMINUS
```

Espressioni logiche (o booleane)

- Un ragionamento del tutto analogo può essere fatto con le espressioni logiche
- Ci sono poche differenze
- Le uniche costanti sono i valori logici `true` e `false` e dunque non é necessario prevedere un token specifico con *valore semantico* (equivalentemente a `num`)
- In termini di priorità, come è noto, `and` ha la precedenza su `or` e l'operatore unario `not` ha la precedenza su tutti

Funzioni esterne

- Un ragionamento del tutto analogo può essere fatto con le espressioni logiche
- Ci sono poche differenze
- Le uniche costanti sono i valori logici `true` e `false` e dunque non é necessario prevedere un token specifico con *valore semantico* (equivalentemente a `num`)
- In termini di priorità, come è noto, `and` ha la precedenza su `or` e l'operatore unario `not` ha la precedenza su tutti

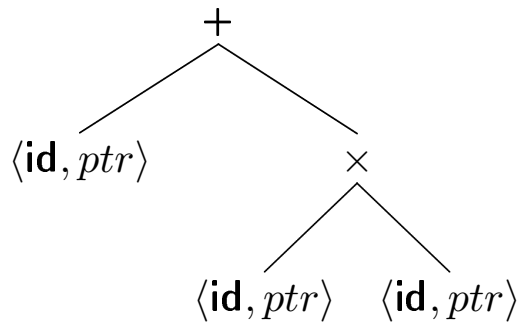
Compilatori

Abstract syntax tree

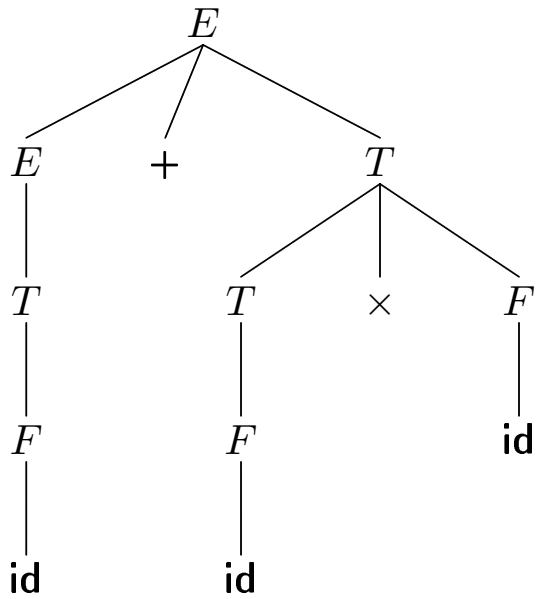
- Un *Abstract Syntax Tree* (*AST*) per un linguaggio L è un albero radicato (*rooted tree*) in cui:
  - i nodi interni rappresentano costrutti di L;
  - i figli di un nodo che rappresenta un costrutto $\mathcal{C}$ rappresentano a loro volta le "componenti significative" di $\mathcal{C}$;
  - le foglie sono "costrutti elementari" (non ulteriormente decomponibili) caratterizzati da un *valore lessicale* (tipicamente un numero o un puntatore alla symbol table).
- Per un linguaggio funzionale, la rappresentazione di un programma mediante AST è particolarmente intuitiva: un sottoalbero di radice f rappresenta infatti in modo naturale l'applicazione di f alle espressioni rappresentate dai suoi sottoalberi
- Come vedremo, e contrariamente a quanto si potrebbe ritenere, anche i costrutti di un linguaggio imperativo possono però essere descritti mediante un AST

Esempio

- Un abstract syntax tree per la frase $\mathbf{id} + \mathbf{id} \times \mathbf{id}$ è

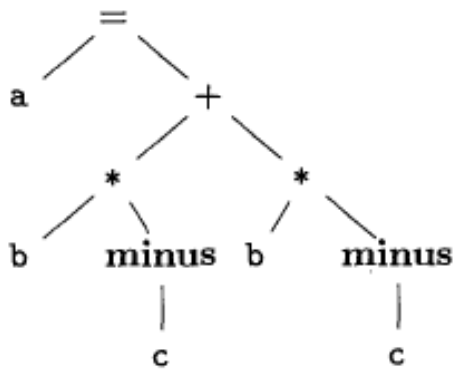


- Abstract syntax tree e parse tree sono oggetti diversi.



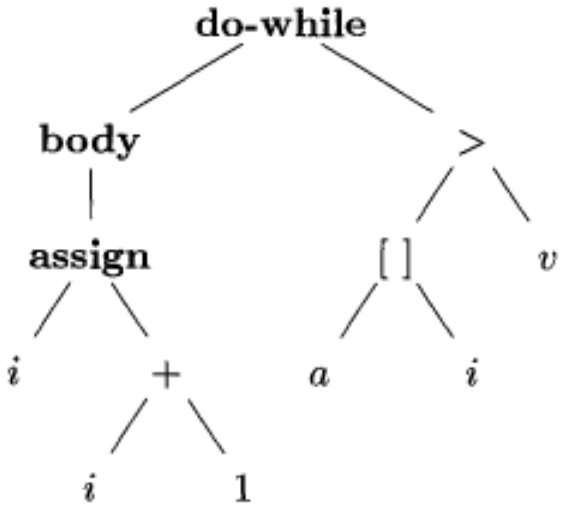
Esempio

- Un AST per l'assegnamento: `a=b*(-c)+b*(-c)`:



Un AST per il comando: **do** `i=i+1` **while** (`a[i]>v`)

·cm



Abstract Syntax Tree

- L'utilità degli AST è riassumibile nelle seguenti affermazioni.
  - Partendo da un AST la generazione di codice lineare è un esercizio "sufficientemente" semplice (anche se l'intero processo è meno efficiente della generazione diretta di codice intermedio).
  - Nella realizzazione di semplici linguaggi intepretati (o comunque di applicazioni dove l'efficienza non sia il principale requisito) gli AST possono rappresentare il risultato ultimo della compilazione.
  - Risulta infatti agevole implementare un software per interpretare gli AST, soprattutto in relazione alla complessità di realizzare un compilatore completo,
- Noi procederemo quindi generando l'AST durante la fase parsing e successivamente "visitando" l'AST per generare il codice intermedio lineare

Un semplice interprete per le espressioni aritmetiche

- La diapositiva seguente presenta lo pseudo-codice per un semplice interprete di AST che rappresenta espressioni aritmetiche.
- Ogni nodo dell'albero tre campi:
  - un campo etichetta (`label`) che, se il nodo è interno, contiene un codice di operatore (come, ad esempio, `PLUS`, `TIMES`, `MINUS`, `UNARY_MINUS`, `...`), se invece il nodo è una foglia contiene un puntatore alla symbol table;
  - un campo puntatore al primo operando (`left`);
  - un campo puntatore all'eventuale secondo operando.
- Lo pseudocodice usa una routine (`apply`) che restituisce il valore dell'applicazione di un operatore binario a due operandi passati come parametri.

Un semplice interprete per le espressioni aritmetiche

EVAL(NODE v)

1: **if** left(v) $\neq$ **nil then**
2: $x \leftarrow$ eval(left(v))
3: **if** label(v) = UNARY_MINUS **then**
4: return $-x$
5: $y \leftarrow$ eval(right(v))
6: return apply(label(v), x, y)
7: **else**
8: return symlookup(label(v))

Architettura del parser

- Il software per la costruzione dell'AST di un programma *LFM* è composto da 5 file
  - `astparser.yy`, il file con la grammatica e i frammenti di codice per la creazione dell'AST
  - `astscanner.ll`, il file che implementa il riconoscimento dei token
  - `astdriver.hh`, il file con la sola definizione delle classi con cui vengono implementati i vari nodi dell'AST
  - `astdriver.cpp`, il file con l'implementazione delle classi
  - `astgen.cpp`, il file con il main program
- Il passaggio delle informazioni fra i vari componenti (main, lexer e scanner) è implementato tramite una classe `driver`