Linguaggi formali e compilazione

Corso di Laurea in Informatica

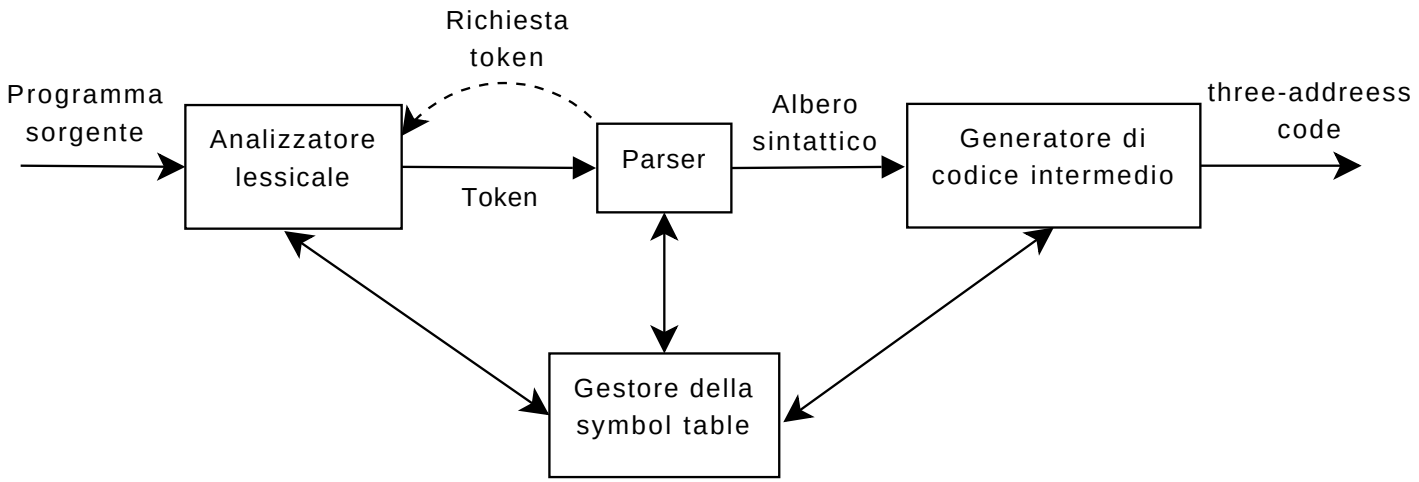
A.A. 2012/2013

Linguaggi formali e compilazione

Cenni su analisi semantica e rappresentazioni intermedie

- Symbol table
- Rappresentazioni intermedie
- Generazione del three-address code

Ricordiamo l'architettura del front-end



Symbol table

- Come evidenziato dalla struttura del front-end, l'analizzatore lessicale presenta al parser una funzione, `get-next-token()` (`yylex` in Lex), che il parser invoca quando deve leggere il prossimo simbolo terminale della grammatica.
- Il risultato esplicito della `get-next-token()` è il nome del prossimo token.
- Il token name è tutto ciò che serve nel caso keyword, operatori, parentesi e simboli di separazione.
- Nei casi in cui il valore di un token è significativo (ad esempio, per i numeri e gli identificatori), l'analizzatore lessicale restituisce tipicamente anche un puntatore alla symbol table.

Symbol table (2)

- In pratica, la funzione `get-next-token()` può restituire una coppia $\langle$nome, valore$\rangle$, oppure il valore viene restituito attraverso una variabile globale (`yylval` nel caso di Lex e Yacc).
- Le informazioni raccolte nella symbol table sono fondamentali nella generazione del codice, ma non solo.
- Mediante tali informazioni il front-end può anche completare l'analisi statica di correttezza del programma.
- Sappiamo infatti che alcune caratteristiche dei linguaggi di programmazione non sono context-free, come ad esempio il vincolo che le variabili siano dichiarate prima dell'uso.

Symbol table (3)

- L'approccio consiste nell'ignorare tale requisito a livello grammaticale, perché il parsing diverrebbe troppo complesso, ma di verificarne il soddisfacimento in una fase di analisi che viene detta *analisi semantica*.
- Ad esempio, in C/C++ la frase `x = y + 1` viene sempre considerata corretta dal punto di vista della grammatica (context free) che definisce il linguaggio
- Essa può tuttavia essere errata in base all'analisi semantica se, poniamo, `y` non è definito ovvero ha un tipo non compatibile con il tipo di `x`.

Symbol table (4)

- Allo scopo, quando incontra un'istruzione che "usa" un identificatore, il parser verifica se esso è già presente nella symbol table (ed eventualmente con quale tipo è stato dichiarato).
- Se non è presente (ovvero se il tipo non è quello "giusto") il parser segnala una condizione di errore di natura *semantica*.
- Un altro esempio di informazione semantica raccolta nella symbol table riguarda numero e tipo dei parametri (ordinati) di una funzione.
- Questa informazione consente di verificare la concordanza fra numero di parametri formali nella definizione di una funzione e argomenti nella "chiamata" (altra caratteristica non context-free).

Static scoping

- Un'accurata realizzazione della symbol table consente anche di implementare le regole statiche di *ambiente* o di visibilità dei simboli (in inglese *static scoping rule*).
- Lo scope (ambiente) della dichiarazione di una variabile è la porzione di programma in cui tale variabile è visibile (e dunque utilizzabile).
- Lo stesso identificatore può essere definito più volte in un programma, ma ad esso saranno associati ambienti diversi.
- Nei moderni linguaggi di programmazione l'ambiente di una variabile è "quasi sempre" determinabile in modo statico, cioè guardando il testo del programma.
- In particolare, la visibilità è determinata dalla struttura di annidamento dei blocchi di programma.
- In questo caso si parla di *static* (o *lexical*) *scoping*.

Esempio (in Pascal)

```
Program P;
var i:integer;
    c:char;
    x:real;
    function f(x:integer);
    var i:integer;
        procedure z;
        var x:integer;
        begin
         (1)
        end
    begin
     (2)
    end
begin
 (3)
end
```

Static scoping (2)

- L'implementazione della nozione di ambiente può essere realizzata nel compilatore mediante una struttura a stack della symbol table.
- Più precisamente, si utilizzano più tabelle concatenate e la symbol table nel suo insieme è una lista di tabelle.
- Quando incontra l'inizio di un blocco di programma, il compilatore inizializza una tabella e la inserisce in testa alla lista.
- Quando incontra la fine di un blocco rimuove la tabella in testa alla lista.

Static scoping (3)

- L'inserimento di un simbolo avviene sempre nella tabella di testa.
- Il look up della symbol table avviene dapprima nella tabella di testa.
- In caso di search hit, il look up termina, altrimenti procede nella successiva tabella.
- Se il simbolo non viene trovato in alcuna tabella si ha una search miss (con eventuale generazione di un messaggio di errore).

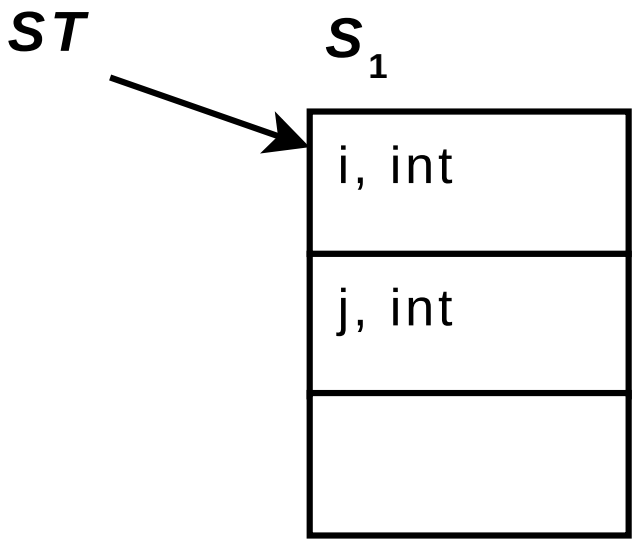
Esempio

- Si consideri, come esempio, il seguente semplice frammento di programma C++:

```
1.  { int i,j;
2.    cin » i » j;
3.    { int i; float x;
4.      i=1;
5.      x=2.0*j;
6.      cout « i « ": " « x;
7.    }
8.    cout « i « ": " « j;
9.  }
```

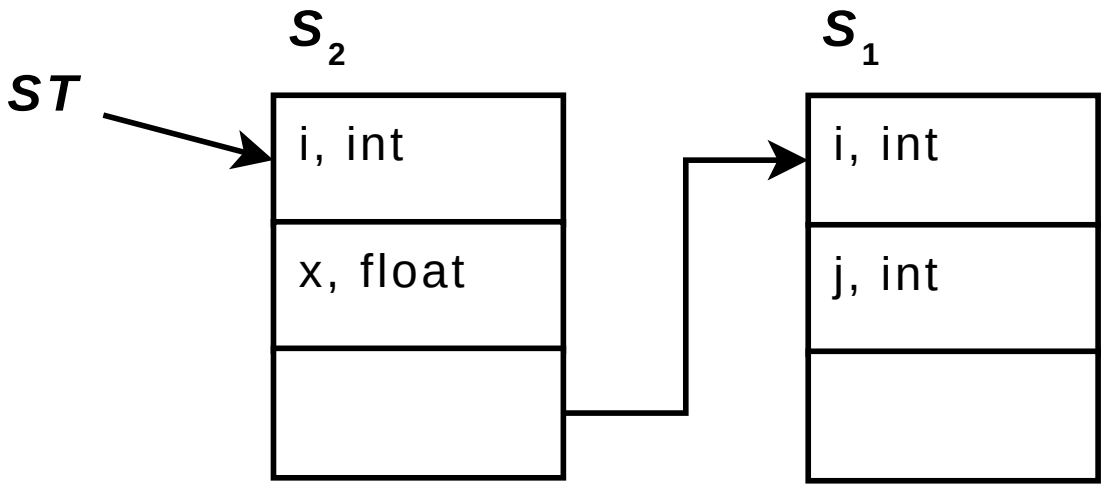
Esempio (continua)

- All'ingresso del blocco più esterno viene creata una tabella S_1 (essenzialmente un dizionario) che diviene la testa della *symbol table*.
- Gli identificatori `i` e `j` di riga 1 vengono inseriti in S_1.



Esempio (continua)

- All'ingresso del blocco interno (riga 3) viene creata (e inserita in testa alla *symbol table*) una seconda tabella, S_2, nella quale vengono poi inseriti gli identificatori `i` e `x` di riga 3 (col loro tipo).



- Il riferimento alla variabile `i` di riga 4 viene risolto con un lookup alla tabella di testa (S_2), che contiene un'entry con chiave `i`.

Esempio (continua)

- Il riferimento alla variabile `j` di riga 5 viene risolto con un lookup alla tabella S_1, dopo aver cercato "inutilmente" in S_2 un'entry con chiave `j`.
- Si noti come la definizione della variabile `i` di riga 3, unitamente al processo di lookup, renda invisibile, nel blocco interno, la variabile `i` definita a riga 1.
- All'uscita del blocco interno, il puntatore alla testa della *symbol table* viene fatto avanzare, con l'effetto di rendere inaccessibile la tabella di testa (S_2).
- Il riferimento alla variabile `i` di riga 8 verrà quindi nuovamente risolto con un lookup a S_1.

Symbol table (5)

- Come già sottolineato, oltre alla implementazione delle regole d'ambiente e a verifiche di correttezza semantica, le informazioni contenute nella symbol table sono fondamentali in sede di generazione del codice
- Ad esempio, la entry per un identificatore contiene:
  - la sequenza di caratteri che ne costituisce il lessema (che può essere usata come chiave per l'accesso alle tabelle);
  - il tipo della variabile;
  - l'indirizzo (relativo) di memoria della variabile.
- Fra gli altri usi, il tipo è importante per stabilire l'occupazione di memoria mentre l'indirizzo serve per generare gli operandi nelle istruzioni macchina.

Le due parti del compilatore

- Il passaggio dal codice sorgente ad un codice macchina efficiente viene tipicamente spezzato in due parti.
  - La prima parte, gestita dal front-end del compilatore, termina con la generazione di un opportuno codice intermedio ed è chiaramente dominata dalle caratteristiche del linguaggio sorgente.
  - La seconda parte, gestita dal back-end, è invece specializzata ad ottenere un codice macchina efficiente in funzione della particolare architettura.
- La suddivisione netta del progetto di un compilatore in front-end e back-end (la prima indipendente dall'architettura e la seconda indipendente dal linguaggio) ha, fra le altre proprietà, un notevole impatto in termini di economicità di progetto.
- Le rappresentazioni intermedie più importanti sono i *syntax tree* e il cosiddetto *three-address code*.

Three address code

- Il codice a tre indirizzi è una rappresentazione intermedia lineare del programma sorgente indipendente da ogni specifica architettura.
- Più precisamente, il three address code è il codice per una architettura astratta di calcolatore.
- Tale modello descrive abbastanza fedelmente la struttura di ogni macchina reale, evitando tuttavia di trattare tutti i complessi dettagli delle moderne achitetture.

Il modello di calcolo per il three address code

- Il calcolatore astratto è in grado di eseguire semplici istruzioni caratterizzate da un *codice operativo* e da *al più* tre indirizzi per gli operandi (da qui il nome).
- Fra le istruzioni disponibili in tale modello di calcolatore, troviamo le operazioni logico/aritmetiche binarie e unarie, le istruzioni di salto (condizionato o incondizionato), il semplice assegnamento, la chiamata di procedura e la gestione di array lineari.
- Ogni istruzione può essere preceduta da una o più etichette (stringhe letterali), utilizzate nelle istruzioni di salto.

Esempio (dal libro di testo)

- L'istruzione condizionale

```
if( x < 100 || x > 200 && x != y ) x = 0;
```

potrebbe essere tradotta nel seguente frammento di three address code:

```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

- Come si vede dall'esempio, il codice è sufficientemente vicino ad un ragionevole codice macchina, pur essendo indipendente da ogni particolare architettura.

Three address code (2)

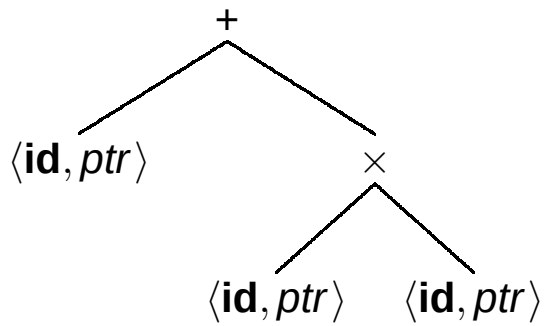
- I moderni compilatori generano codice intermedio lineare (come il three address code, del quale torneremo ad occuparci più avanti) in maniera diretta.
- In queste note (per ragioni didattiche) supporremo invece che il three address code sia il risultato finale di una serie di passaggi “più fini”.
- Tali ulteriori passaggi intermedi trasformano il programma sorgente in rappresentazioni ad allbero equivalenti.
- Queste rappresentazioni sono lo stesso parse tree e, soprattutto, l’abstract syntax tree.
- Peraltro, la generazione esplicita del parse tree è (quasi sempre) evitabile.

Abstract syntax tree

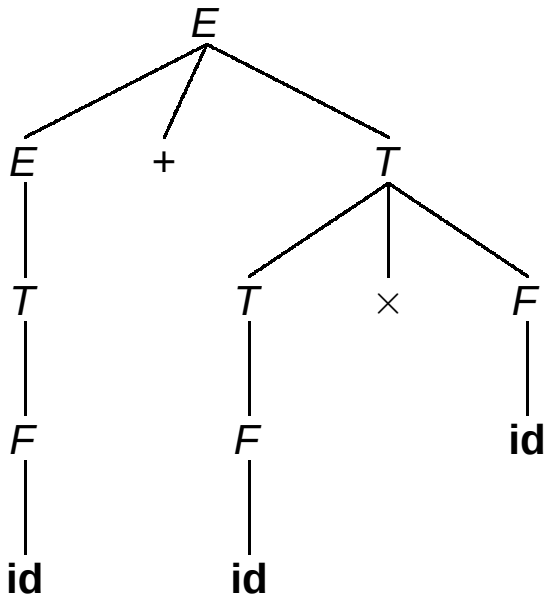
- Un *abstract syntax tree* (*AST*) per un linguaggio *L* è un albero in cui:
  - i nodi interni rappresentano costrutti di *L*;
  - i figli di un nodo che rappresenta un costrutto $\mathcal{C}$ rappresentano a loro volta le “componenti significative” di $\mathcal{C}$;
  - le foglie sono “costrutti elementari” (non ulteriormente decomponibili) caratterizzati da un *valore lessicale* (tipicamente un numero o un puntatore alla symbol table).
- Le diapositive seguenti illustrano la nozione di abstract syntax tree.

Esempio

- Un abstract syntax tree per la frase **id** $+$ **id** $\times$ **id** è

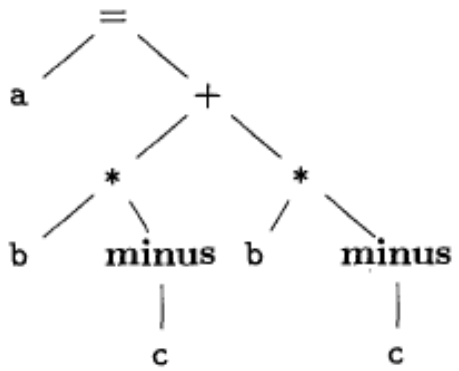


- Abstract syntax tree e parse tree sono oggetti diversi.



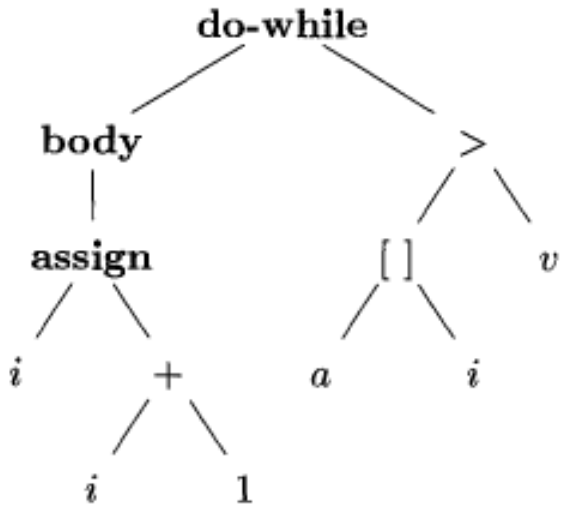
Esempio

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



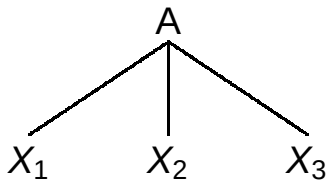
- Un AST per il comando

do `i=i+1` **while** (`a[i]>v`)

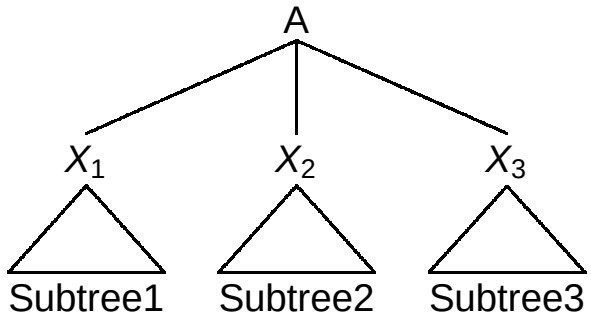


Generazione di un AST

- La generazione esplicita di un parse tree può essere ottenuta agevolmente:
  - in un parser top-down, associando ad ogni produzione (ad esempio) $A \rightarrow X_1 X_2 X_3$ opportuno codice per la costruzione della porzione di albero



  - in un parser bottom-up, associando ad ogni riduzione (ad esempio) $X \rightarrow X_1 X_2 X_3$ opportuno codice per la costruzione della porzione di albero



- Quando è disponibile il parse tree, la generazione del corrispondente AST può essere ottenuta "essenzialmente" mediante un algoritmo di visita.

Abstract syntax tree (2)

- L'utilità degli AST è riassumibile nelle seguenti affermazioni.
  - Partendo da un AST la generazione del three address code 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 molto semplice (in generale, e in rapporto alla complessità di realizzare un compilatore completo) implementare un software per interpretare gli AST.

Assunzioni

- In quest'ultima parte del corso presentiamo di alcuni semplici esempi di generazione di codice intermedio.
- Non intendiamo presentare una descrizione completa, sia per ragioni di tempo sia perché (a nostro avviso) questa attività è molto meno interessante dal punto di vista delle idee sottostanti.
- Gli esempi hanno dunque il solo scopo di illustrare l'approccio base per la generazione di codice intermedio, ignorando quegli aspetti che richiederebbero un ragionamento più approfondito.
- Faremo inoltre tre ipotesi molto forti, e precisamente: (1) di disporre dell'abstract syntax tree delle stringhe da tradurre; (2) che il risultato debba essere presentato sotto forma di file alfanumerico, (3) che esista un unico tipo di dato nei programmi sorgenti (ad esempio, numeri interi).

Istruzioni specifiche del three-address code

- Allo scopo di presentare gli esempi, considereremo solo le seguenti istruzioni, il cui significato dovrebbe risultare chiaro:
  - $A \leftarrow B$ `op` C, dove `op` $\in \{+, -, \times, /\}$ e A, B e C sono identificatori definiti dall'utente nel programma sorgente oppure temporanee generate dal parser;
  - $A \leftarrow B$, dove A e B sono definiti come al punto precedente;
  - $A \leftarrow$ `op` B, dove `op` è un operatore unario;
  - `goto` L, dove L è un'etichetta generata dal parser;
  - `if` A `goto` L, dove A è un identificatore definito dall'utente oppure una temporanea generata dal parser ed L è un'etichetta generata dal parser;
  - `ifFalse` A `goto` L, dove A ed L sono definiti come al punto precedente.

Ulteriori assunzioni

- Ipotizzeremo che il parser possa invocare una funzione per generare identificatori univoci, come pure una funzione per generare etichette univoche.
- Assumeremo inoltre la disponibilità di una funzione, che chiameremo *emit*(), che stampa la stringa passata come parametro (che rappresenta una porzione del programma in three-address code) su un opportuno supporto di output.
- Assumeremo infine che il generico nodo dell'abstract syntax tree abbia la seguente struttura:
  - un campo `label` che caratterizza il tipo di nodo (assegnamento, operatore, comando if, ...);
  - se significativo (ad esempio nel caso di identificatore o operatore), un puntatore alla symbol table per il corrispondente valore lessicale, accessibie mediante la funzione *lexval*;
  - uno o più puntatori ai nodi che rappresentano i componenti del costrutto, accessibili mediante i campi `c1`, `c2`, ...

La struttura generale del traduttore

- Con le ipotesi fatte, il traduttore (da syntax tree a three-address code) può essere organizzato come procedura ricorsiva il cui corpo è costituito essenzialmente da una "grossa" istruzione `case` (o, se si preferisce, `switch`), che analizza il nodo p passato come parametro e, a seconda del tipo, esegue operazioni diverse.
- Data la struttura del parse tree, la generazione del codice per un dato nodo implicherà poi una o più chiamate ricorsive per la generazione del codice associato ai nodi figli.
- La procedura, che chiameremo *gencode*, riceve un ulteriore parametro (*RES*) che è una stringa (eventualmente vuota) alla quale (vista come nome di variabile) deve essere assegnato il risultato calcolato dal codice generato per il nodo *p*.

Procedura gencode

Procedure 1 *gencode*(*string* RES, *AST** p)

```
tag ← (p → label)
case tag of
    id :    ...
    number :    ...
    assignment :    ...
    comparison :    ...
    binaryop :    ...
    unaryminus :    ...
    seq :    ...
    if :    ...
    ifElse :    ...
    while :    ...
...
    default :    error()
end
```

Il caso degli identificatori

- Si tratta del caso più semplice da trattare.
- Infatti, se un nodo è etichettato come identificatore, tutto ciò che bisogna fare è semplicemente emettere una stringa che ne rappresenta il valore lessicale.
- Al riguardo, utilizziamo una funzione *toString* (esiste anche in Java), che, nel caso il valore lessicale dell'identificatore sia già internamente rappresentato come stringa, equivale ad un no-op.
- Per altri tipi di nodo, *toString* svolge effettivamente una funzione: ad esempio, se il nodo è un operatore binario, la chiamata $toString(lexval(p))$ ne fornisce la consueta rappresentazione matematica (+,-, ecc).

Il caso degli identificatori (continua)

- Il codice relativo a questo caso è dunque:

  id : `string name` $\leftarrow$ *toString*(*lexval*(*p*))
  if not *empty*(`RES`) **then**
  emit(`RES`+“$\leftarrow$”+`name`)
  else
  emit(`name`)

  dove l’operatore + applicato a stringhe denota concatenazione.
- Si noti anche il controllo (che sarà ricorrente anche nei seguenti esempi) sulla stringa `RES`.
- Se `RES` non è la stringa vuota, il codice da generare deve infatti prevedere un assegnamento al nome da essa rappresentato.

Costanti numeriche

- Il caso delle costanti numeriche è identico a quello degli identificatori.
- C'è solo un maggior lavoro (nascosto) da parte della procedura `toString`, che deve ri-trasformare in sequenza ASCII un numero rappresentato internamente in virgola fissa o virgola mobile.

id : `string const` $\leftarrow$ *toString*(*lexval*(p))
if not *empty*(`RES`) **then**
    emit(`RES`+"$\leftarrow$"+`const`)
else
    emit(`const`)

Assegnamento

- Un nodo etichettato come *assignment* ha due figli, il primo dei quali deve necessariamente essere un *id*.
- Il codice da generare prevede la chiamata ricorsiva al secondo figlio, in modo che lasci il valore nella variabile il cui nome è il valore lessicale del primo figlio.
- In altre parole:

assignment : `string name` $\leftarrow$ *lexval*(`p` $\rightarrow$ `c1`)
if not *empty*(`RES`) **then**
    gencode(`name`, `p` $\rightarrow$ `c2`)
    emit(`RES`+"$\leftarrow$"+*name*)
else
    gencode(`name`, `p` $\rightarrow$ `c2`)

Gli operatori (aritmetici) binari

- Se l'etichetta del nodo è un operatore binario bisogna:
  - generare ricorsivamente il codice per il figlio di sinistra, in modo che lasci il risultato in una variabile il cui nome univoco è generato dal parser stesso (supporremo che tali nomi abbiano la forma `TEMP`*n*, con *n* progressivo);
  - generare (analogamente) il codice per figlio di destra, in modo che lasci il risultato in una seconda variabile;
  - generare la stringa per un'istruzione a tre o due indirizzi (a seconda del valore del parametro `RES`) che esegue l'operazione indicata dal particolare operatore binario sui dati memorizzati nelle temporanee.

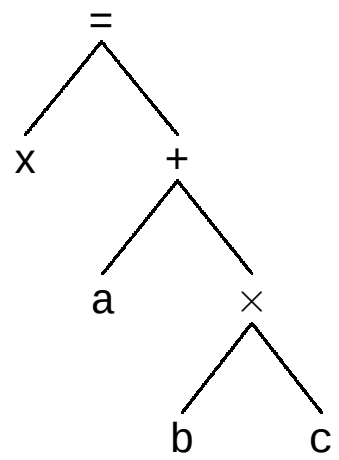
Gli operatori (aritmetici) binari (continua)

- Il codice corrispondente è:

  binaryop :
  `string t1` $\leftarrow$ *new temporary*()
  `string t2` $\leftarrow$ *new temporary*()
  gencode(`t1`, `p` $\rightarrow$ `c1`)
  gencode(`t2`, `p` $\rightarrow$ `c2`)
  `string op` $\leftarrow$ *toString*(*lexval*(*p*))
  if not *empty*(`RES`) **then**
  emit(`RES`+"$\leftarrow$"+`t1` + `op` + `t2`)
  else
  emit(`t1` + `op` + `t2`)

Esempio

- Per l'abstract syntax tree del comando C/C++ $x = a + b \times c$ (rappresentato in modo da evidenziare direttamente il valore lessicale degli operatori e degli identificatori)



viene generato il seguente codice a tre indirizzi

$\texttt{temp1} \leftarrow \texttt{a}$
$\texttt{temp3} \leftarrow \texttt{b}$
$\texttt{temp4} \leftarrow \texttt{c}$
$\texttt{temp2} \leftarrow \texttt{temp3} \times \texttt{temp4}$
$\texttt{x} \leftarrow \texttt{temp1} + \texttt{temp2}$

L'operatore "meno" unario

- È semplicissimo da realizzare.
- Si tratta dapprima di generare il codice per l'espressione che costituisce l'unico figlio, lasciando il risultato il una temporanea.
- Al risultato si applica poi l'operatore meno unario `uminus`.
- Il codice è:

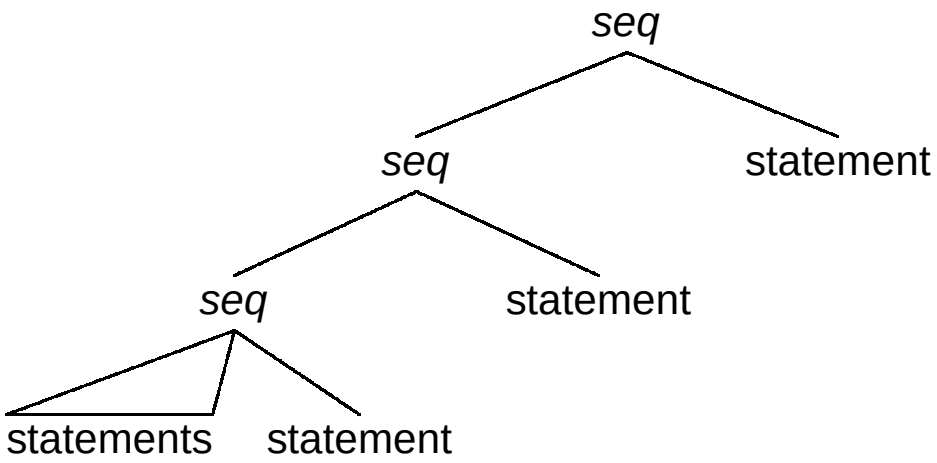
unaryminus :
`string t` $\leftarrow$ *new temporary*()
gencode(`t`, `p` $\rightarrow$ `c1`)
if not *empty*(`RES`) **then**
  emit(`RES`+"$\leftarrow$ `uminus`"+`t`)
else
  emit("`uminus` + `t`″)

Sequenza

- Una sequenza di comandi, definita dalle produzioni

$$L \rightarrow L; S \mid S,$$

produce alberi sintattici con la struttura indicata di seguito (in cui ogni singolo statement può, a sua volta, essere un syntax tree)

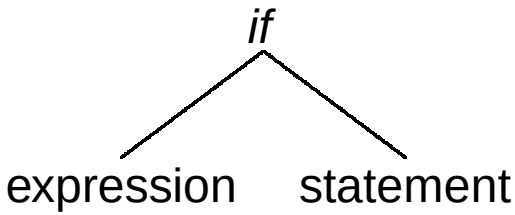


- Il codice corrispondente consiste semplicemente di due chiamate ricorsive:

seq :
gencode(“”, p $\rightarrow$ c1)
gencode(“”, p $\rightarrow$ c2)

Comando “If then”

- È rappresentato da alberi sintattici del tipo



- I passi per effettuare la traduzione sono i seguenti:
  - si genera ricorsivamente il codice per calcolare l’espressione, lasciando il risultato in una temporanea;
  - si genera un’etichetta e si emette un’istruzione di salto a tale etichetta, condizionato al valore falso della temporanea;
  - si genera quindi ricorsivamente il codice per il comando (che verrà dunque eseguito se il valore della temporanea è vero);
  - infine si emette l’etichetta generata (che andrà ad etichettare la prossima istruzione a tre indirizzi, non emessa dal trattamento del condizionale).

Comando “If then” (2)

- Il codice corrispondente è:

  if :
  `string t` $\leftarrow$ *new temporary*()
  gencode(`t`, `p` $\rightarrow$ `c1`)
  `string l` $\leftarrow$ *new label*()
  emit(“`ifFalse `”+`t`+“ `goto `”+`l`)
  gencode(“”, `p` $\rightarrow$ `c2`)
  emit(`l`+“: ”)

Comando “If then else”

- È solo leggermente più complicato del caso precedente, per cui presentiamo direttamente il codice

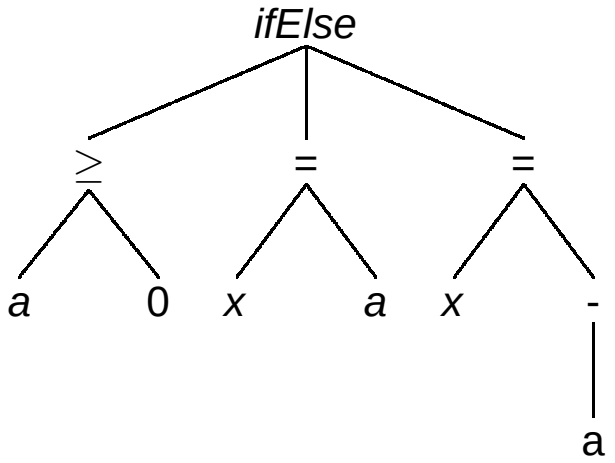
  ifElse :
  `string t` $\leftarrow$ *new temporary*()
  gencode(`t`, `p` $\rightarrow$ `c1`)
  `string l1` $\leftarrow$ *new label*()
  emit(“`ifFalse `”+`t`+“ `goto `”+`l1`)
  gencode(“”, `p` $\rightarrow$ `c2`)
  `string l2` $\leftarrow$ *new label*()
  emit(“`goto `”+`l2`)
  emit(`l1` + “ :”)
  gencode(“”, `p` $\rightarrow$ `c3`)
  emit(`l2`+“: ”)

Esempio

- Alla frase C/C++

$$\texttt{if}\ a >= 0\ \texttt{then}\ x = a\ \text{else}\ x = -a$$

corrisponde il seguente abstract syntax tree



(si ricordi che abbiamo scelto di evidenziare direttamente il valore lessicale degli operatori e degli identificatori anziché inserire simboli astratti e riferimenti alla symbol table).

Esempio (continua)

- Il codice a tre indirizzi corrispondente è:

```
temp2 ← a
temp3 ← 0
temp1 ← temp2 ≥ temp3
ifFalse temp1 goto label1
x ← a
goto label2
label1 : temp4 ← a
x ← −temp4
label2 :
```

Comando “while”

- Come ultimo caso, consideriamo la traduzione di abstract syntax tree corrispondenti al costrutto while.
- Il costrutto ha due componenti, la condizione e lo statement da ripetere finché la condizione è vera.
- La strategia di traduzione consiste quindi nel generare il codice per la condizione, emettere un’istruzione di salto condizionato (`ifFalse`), generare il codice per il comando ed emettere un’istruzione di salto incondizionato al codice generato per la condizione.
- Lo pseudocodice dettagliato è riportato nella successiva trasparenza.

Comando “while” (2)

while :
`string t` $\leftarrow$ *new temporary*()
`string l1` $\leftarrow$ *new label*()
emit(`l1`+“: ”)
gencode(`t`, `p` $\rightarrow$ `c1`)
`string l2` $\leftarrow$ *new label*()
emit(“`ifFalse `”+`t`+“ `goto `”+`l2`)
gencode(“”, `p` $\rightarrow$ `c2`)
emit(“`goto `”+`l1`)
emit(`l2`+“: ”)