Linguaggi e compilatori

Corso di Laurea in Informatica

Mauro Leoncini

A.A. 2023/2024

Linguaggi e compilatori

Che cosa è Bison

- `Bison` è una versione moderna di `Yacc`, ovvero uno strumento per generare parser *LR* nei linguaggi `C` o `C++`
- Lo stesso nome `Yacc` è acronimo di *Yet another compiler compiler*
- L'utilizzo per il `C++` è sensibilmente più complesso rispetto al'utilizzo un `C` puro
- Ci limiteremo quindi a presentarne gli elementi essenziali con la messa a punto di un *template* che possa essere "riciclato" per diversi utilizzi

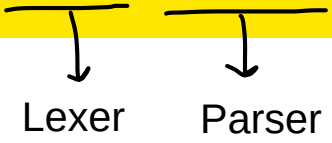
Interazione Flex-Bison

- `Bison` opera su un flusso di token che può essere generato da un opportuno programma custom ma che, più generalmente e "comodamente", può essere prodotto da `Flex`
- Ricordiamo brevemente come funziona `Flex`
- Il programmatore prepara un file (di solito con estensione `l` o `ll`) il cui contenuto fondamentale è rappresentato da:
  1. la definizione dei token da riconoscere, fornita mediante espressioni regolari, e
  2. i frammenti di codice che devono essere eseguiti nel momento in cui i vari tipi di token vengono riconosciuti
- Questo *sorgente Lex* viene poi interpretato da `Flex` che produce in output codice `C` o `C++` compilabile

Interazione Flex-Bison

- Il funzionamento di `Bison` è analogo
- Il programmatore prepara un file (di solito con estensione `y` o `yy`) in cui al programmatore è richiesto essenzialmente di specificare:
  1. i nomi dei *token* e i simboli *nonterminali* della grammatica, indicando in particolare quale sia l'assioma
  2. le produzioni della grammatica
  3. i frammenti di codice che devono essere eseguiti nel momento in cui il parser effettua una *riduzione*
- Bison interpreta il *sorgente Yacc* producendo il codice `C/C++`
- Quando l'applicazione prevede sia scanning che parsing (ad esempio, quando si scrive un compilatore...) è necessario definire con precisione l'interazione fra i codici prodotti da `Flex` e da `Bison`

Interazione Flex-Bison



- Al netto di non pochi dettagli e direttive di controllo, il procedimento richiede la corretta implementazione di tre "meccanismi" fondamentali:
  1. la definizione dei token e la loro condivisione fra `Flex` e `Bison`;
  2. il flusso ordinato dei token da `Flex` a `Bison`;
  3. la condivisione delle informazioni da mostrare in caso di errori.
- Il programmatore deve inserire le informazioni pertinenti nei due file sorgenti, che chiameremo `scanner.ll` e `parser.yy` (o semplicemente `scanner` e `parser`).

Definizione dei token

Token type/Token name -> NUMBER , For
Tipo -> int,float,char...

- I nomi dei token vengono definiti nel (file) `parser` mediante la *direttiva* `%token`
- Il nome di un token ne specifica anche il tipo, ed infatti viene anche detto *token type*
- A seconda del tipo, sappiamo che un token può essere caratterizzato anche da un *valore*, detto *semantic* o *lexical value*, che è un "oggetto C/C++"
- Entra quindi in gioco un secondo concetto di tipo, precisamente il tipo di tale oggetto C/C++, che non va dunque confuso con il *token type*
- Ad esempio, un token di tipo numerico può avere token name/type `NUMBER` mentre il tipo del suo valore in quanto oggetto C++ potrebbe essere `int` o `float`
- Per evitare confusione, preferiamo usare la locuzione *token name*, piuttosto che *token type*

Definizione dei token

- La direttiva `%token` può dunque assumere due forme differenti, come mostrato dai due esempi seguenti

```
%token PLUS "+"
%token<float> NUMBER "number"
```

→ Token name

→ Usato nelle produzioni

- In entrambi i casi il *token name*, utilizzato dallo scanner per indicare i lessemi riconosciuti nel file di input, è indicato dal nome scritto in maiuscolo (`PLUS` o `NUMBER`)
- La stringa che segue, in questo caso scritta in minuscolo, definisce il modo con cui i token vengono indicati come *simboli terminali* nelle produzioni che seguiranno
- Le definizioni dei token presenti nel `parser` vengono scritte da Bison in un *header file* (es. `parser.hh`) generato durante il processo di compilazione
- Chiaramente, tale header file deve essere incluso nello `scanner`

Flusso da scanner a parser

- L'implementazione del flusso ordinato di token richiede a sua volta che siano definiti due aspetti importanti
- Innanzitutto è necessario che il parser "sappia" come invocare lo scanner per richiedere i token
- È cioè necessario che il parser conosca il *prototipo* della funzione `yylex` che costituisce il punto di ingresso dello scanner
- Nei primi esempi di uso di `Lex` istanziavamo un oggetto (di una classe derivata) della classe `FlexLex` e poi ne invocavamo il metodo `yylex`

```
FlexLexer* lexer = new yyFlexLexer;
lexer->yylex();
```

- Ci domandiamo se possiamo procedere ancora nello stesso modo

Flusso da scanner a parser

- Per applicazioni più complesse rispetto a quelle che abbiamo visto finora il meccanismo è necessariamente più articolato
- Il *prototipo* della funzione `yylex` può infatti essere deciso dal programmatore per rendere possibile il passaggio di parametri al lexer
- Il metodo "standard" per prevedere uno specifico prototipo prevede che i file sorgenti di parser e lexer includano una macro, di nome (obbligatorio) `YY_DECL`, che appunto definisca il prototipo
- Ad esempio, con la macro → Permette di personalizzare il metodo yylex()

```
# define YY_DECL \
yy::parser::symbol_type yylex (myclass& myobj)
YY_DECL;
```

Espande la macro

Token name

Miei parametri

si prevede che le chiamate al lexer includano come argomento oggetti della classe `myclass`

Flusso da scanner a parser

- La macro viene tipicamente inserita in un *header file* importato da lexer e parser che tuttavia la usano in modo diverso
- Per il parser, che effettuerà le chiamate, è sufficiente espandere la macro a inizio file in modo da disporre della definizione

```
yy::parser::symbol_type yylex (myclass& myobj);
```

- Nel lexer però la funzione viene dichiarata e questo comporta che il prototipo si trovi immediatamente prima del body
- Nel lexer la macro va dunque espansa proprio "davanti al body" (allo scopo si utilizza il *macro processor* `M4`)
- Nel lexer è poi possibile (mediante istruzioni opportune) inserire codice nel generico template `yylex`; codice che evidentemente usa il parametro previsto nel prototipo (che altrimenti non servirebbe a nulla)

Flusso da scanner a parser

- Il secondo aspetto dell'interazione fra scanner e parser riguarda il modo con lo scanner restituisce i token al parser.
- Si possono avere due soluzioni diverse
- La prima è quella classica relativa a programmi in `C`: lo scanner restituisce un numero intero come *token type* e memorizza il *semantic value* (eventuale) del token in una variabile (`yylval`) il cui tipo è solitamente definito come `union` nella direttiva omonima (`%union`)
- Se si adotta questa modalità lo scanner usa una normale istruzione `return` per restituire il controllo al parser
- Il secondo modo è tipico delle applicazioni realizzate nel linguaggio `C++`, è meno immediato ma fornisce più garanzie riguardo la *type safety*

Flusso da scanner a parser

- Se definisci il token "number" avrai la funzione: make_number()

- Nel parser è necessario usare la direttiva `%define api.token.constructor`
- Con tale direttiva, per ogni token `XXX` definito nel file `parser.yy`, Bison genera una funzione `make_XXX` che lo scanner può utilizzare per restituire un cosiddetto *complete symbol*
- In dipendenza della natura di `XXX`, la funzione `make_XXX` può avere 1 o più parametri in modo da definire completamente il token, mediante la specifica di *token type*, *semantic value* e *location* (per quest'ultima si veda oltre)

Flusso da scanner a parser

- Ad esempio, per un semplice token `PLUS`, lo scanner potrebbe restituire il controllo al parser eseguendo

EX1: `return yy::parser::make_PLUS(location)`

mentre per un token `NUMBER` con semantic value `n` potrebbe eseguire

EX2: `return yy::parser::make_NUMBER(n,location)`

- Il vantaggio è che, con queste funzioni, il controllo sui tipi è più stringente
- Ad esempio, usare

  `return yy::parser::make_NUMBER(n)`

  oppure

  `return yy::parser::make_NUMBER("a string", location)`

  produrrebbe errori in compilazione

Location

- è una classe, gli oggetti di tipo location permette di localizzare i token.
  Begin -> int indice di riga , int indice di colonna
  End -> int indice di riga , int indice di colonna

- In caso di errori, è opportuno che il parser fornisca informazioni utili all'individuazione (e quindi alla correzione) dei medesimi
- Per il tracciamento degli errori, Bison mette a disposizione la classe `location` che permette di "localizzare" i token.
- Ogni location è caratterizzata da due posizioni, chiamate `begin` e `end`
- A sua volta, una posizione è definita (oltre che dal nome del file) da un indice di riga e uno di colonna
- Sulle location sono definiti i seguenti metodi, utilizzabili nello scanner
  - `step()`, che fa avanzare la posizione `begin` fino al valore di `end`
  - `columns(count)`, che fa avanzare l'indice di colonna di `end` di una quantità pari al valore `count` (di regola la lunghezza del lessema riconosciuto)
  - `lines(count)`, che fa avanzare l'indice di riga di `end` del valore `count` (anche in questo caso pari al numero di righe del pattern riconosciuto, tipicamente uno o più `\n`) e pone al valore 1 il corrispondente indice di colonna

Location

- I metodi presentati consentono di tenere traccia della posizione del token corrente, anche alla luce di due ulteriori meccanismi messi a disposizione da `Flex`
  - la possibilità di eseguire codice custom nel momento in cui la funzione `yylex` va in esecuzione;
  - la possibilità di eseguire codice custom nel momento in cui `yylex` riconosce un lessema (qualsiasi esso sia).
- In tale codice custom, utilizzando le funzioni `step`, `columns` e `lines`, lo scanner può tenere aggiornata la posizione del token corrente, posizione che viene "inviata" al parser mediante le già citate funzioni `make_XXX`
- In caso di errore sintattico, il parser è quindi in grado di localizzare il token che ha provocato errore

Un primo esempio (parziale)

- Presentiamo una prima applicazione, una semplice (ma non troppo!) calcolatrice interattiva che opera solo sui numeri interi ma che può fare uso di variabili
- Ci concentreremo soprattutto sui *meccanismi di interazione* fra scanner, parser e il programma principale.
- In un secondo momento dovremo entrare anche nel dettaglio del funzionamento del parser, cioè di come scrivere la grammatica e il codice associato alle produzioni
- Nelle slide seguenti si tenga presente che `yy` è il nome default del *namespace* dove Bison crea diversi elementi dell'applicazione (ad esempio, le classi `parser` e `location`)
- Il nome `calc` si riferisce invece al namespace definito dall'applicazione calcolatrice

Lo scanner per la calcolatrice (codice non completo)

```
id      [a-zA-Z][a-zA-Z_0-9]*
int     [0-9]+
blank   [ \t]
%{
  // Macro definizione inserita nel codice generato da Flex
  // YY_USER_ACTION viene eseguita quando Flex riconosce un token
  # define YY_USER_ACTION  loc.columns (yyleng);
%}
%%
%{
  // Codice eseguito tutte le volte che viene invocata yylex
  yy::location& loc = location;
  loc.step ();
%}
{blank}+    loc.step ();
[\n]+       loc.lines (yyleng); loc.step ();
"-"         return yy::parser::make_MINUS  (loc);
"+"         return yy::parser::make_PLUS   (loc);
"*"         return yy::parser::make_STAR   (loc);
"/"         return yy::parser::make_SLASH  (loc);
"("         return yy::parser::make_LPAREN (loc);
```

Definizioni regolari

Espansa

Fa avanzare END
di una quantità = yyleng

Lo scanner per la calcolatrice

```
")"         return yy::parser::make_RPAREN (loc);
"="         return yy::parser::make_ASSIGN (loc);
{int}       {
  errno = 0;
  long n = strtol (yytext, NULL, 10);
  if (! (INT_MIN <= n && n <= INT_MAX && errno != ERANGE))
  throw yy::parser::syntax_error (loc, "integer is out of range: "
  + std::string(yytext));
  return yy::parser::make_NUMBER (n, loc);
}
{id}    return yy::parser::make_IDENTIFIER (yytext, loc);
.    {
  throw yy::parser::syntax_error
  (loc, "invalid character: " + std::string(yytext));
}
<<EOF>>     return yy::parser::make_END (loc);
%%
```

Il parser per la calcolatrice

```
%code {
    #include "calc++.hpp"
    extern std::map<std::string,int> variables;
    extern calc::location location;
    extern int result;
}

%define api.token.prefix {TOK_}
%token
END   0   "end of file"
ASSIGN    "="
MINUS     "-"
PLUS      "+"
STAR      "*"
SLASH     "/"
LPAREN    "("
RPAREN    ")"
;

%token <std::string> IDENTIFIER "identifier"
%token <int> NUMBER "number"
%nterm  <int> exp
```

Diversi da gli altri token perchè hanno il tipo

Il parser per la calcolatrice

```
%%
%start unit;
unit: assignments exp  { result = $2; };

assignments:
%empty                          {}
| assignments assignment {};

assignment:
"identifier" "=" exp { variables[$1] = $3; };

%left "+" "-";
%left "*" "/";
exp:
   exp "+" exp      { $$ = $1 + $3; }
| exp "-" exp       { $$ = $1 - $3; }
| exp "*" exp       { $$ = $1 * $3; }
| exp "/" exp       { $$ = $1 / $3; }
| "(" exp ")"       { $$ = $2; }
| "identifier"      { $$ = variables[$1]; }
| "number"          { $$ = $1; };
%%
```

$$ = $0

Serve perchè questa grammatica è ambigua. E quindi facendo così dico sia +,-,*,/ sono associative a sinistra. e inoltre *,/ hanno priorità più alta

In questo es: $0 -> exp $1 = exp $2 = "+" $3 = exp

Lo schema è questo
<name> : -> $0
exp + exp * cane + sus
$1 $2 $3 $4 $5 $6 $7

Il main program

```
#include "parser.hpp"

// External declarations (functions defined in scanner.ll)
void scan_begin (const std::string &f);
void scan_end ();

yy::location location;
bool trace_parsing = false;
bool trace_scanning = false;
std::map<std::string,int> variables;
int result;

int parse(const std::string &f) {
   location.initialize (&f);
   scan_begin (f);
   yy::parser parser;
   parser.set_debug_level (trace_parsing);
   int res = parser.parse ();
   scan_end ();
   return res;
}
```

Il main program

```
int main (int argc, char *argv[])
{
    int res = 0;
    for (int i = 1; i < argc; ++i)
        if (argv[i] == std::string ("-p"))
            trace_parsing = true;
        else if (argv[i] == std::string ("-s"))
            trace_scanning = true;
        else if (!parse (argv[i]))
            std::cout << result << '\n';
        else
            res = 1;
    return res;
}
```

Comprendere il parser

- Secondo il formato richiesto da `Bison`, nelle produzioni i terminali sono racchiusi fra doppi apici
- Inoltre, la testa della produzione è separata dal body mediante i due punti (e non la freccia)
- Alla luce di queste precisazioni, possiamo cercare di riscrivere la grammatica utilizzata nell'esempio secondo lo "stile" che abbiamo usato finora nelle trattazioni teoriche
- Riconosciamo immediatamente le produzioni che descrivono la sintassi delle espressioni

$$\begin{aligned} E &\rightarrow E{+}E \mid E{*}E \mid E{-}E \mid E/E \\ E &\rightarrow (E) \mid \texttt{number} \mid \texttt{id} \end{aligned}$$

Comprendere il parser

- Come sappiamo, questa grammatica è ambigua e una derivazione può non rispettare la precedenza degli operatori
- Essa però ha il vantaggio di essere più semplice rispetto a quella (che abbiamo usato più volte) che include non terminali `T` e `F`, ed é caratterizzata da derivazioni in generale decisamente più corte
- Confrontate, ad esempio, la derivazione canonica destra della stringa `number+number` nella grammatica che rispetta le precedenze con la derivazione seguente

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \texttt{number} + E \\ &\Rightarrow \texttt{number} + \texttt{number} \end{aligned}$$

Comprendere il parser

- La maggiore semplicità non sembra però sufficiente a compensare la mancata osservanza delle precedenze (che è ben più grave)
- Bison però mette a disposizione le direttive `%left` e `%right` mediante le quali il programmatore può specificare sia la direzione dell'associatività (sinistra o destra) in modo esplicito, sia le precedenze, in modo implicito mediante l'ordine in cui compaiono le direttive
- Nell'esempio troviamo, nell'ordine

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

- Questo "dice" a Bison che gli operatori sono tutti associativi a sinistra ma che * e / hanno una priorità maggiore, perché sono dichiarati dopo

Comprendere il parser

- Ritornando alla grammatica, vediamo che ci sono altre produzioni

$$\begin{aligned} U &\rightarrow S\,E \\ S &\rightarrow S\,A \mid \epsilon \\ A &\rightarrow \texttt{id} := E \end{aligned}$$

  dove abbiamo usato U per `unit`, S per `assignments`, A per `assignment`
- Abbiamo tutte le conoscenze per capirne il senso!
- Una `unit`, che è l'assioma (indicato dalla direttiva `%start`) è una sequenza di assegnamenti, che può anche essere vuota, seguita da un'epressione, che invece deve essere presente
- A loro volta, gli assegnamenti sono formati da un identificatore, dal simbolo terminale :=, utilizzato al posto del solo = in alcuni linguaggi, e da un'espressione.

Produzioni e codice associato

- Alle produzioni il programmatore può associare codice C/C++
- In tale codice possono apparire anche alcune sequenze speciali, $$, $1, $2, ..., che naturalmente richiedono una spiegazione
- Tali sequenze denotano variabili interne che Bison associa ordinatamente ai simboli che compaiono nella produzione
- $$ indica la variabile associata al non terminale testa della produzione mentre $1, $2, ... sono associate ai simboli (terminali o non teminali) nella parte destra della produzione
- Si noti che, se il programmatore non scrive alcun codice per una data, produzione, Bison inserisce implicitamente la riga:

```
{ $$ = $1; }
```

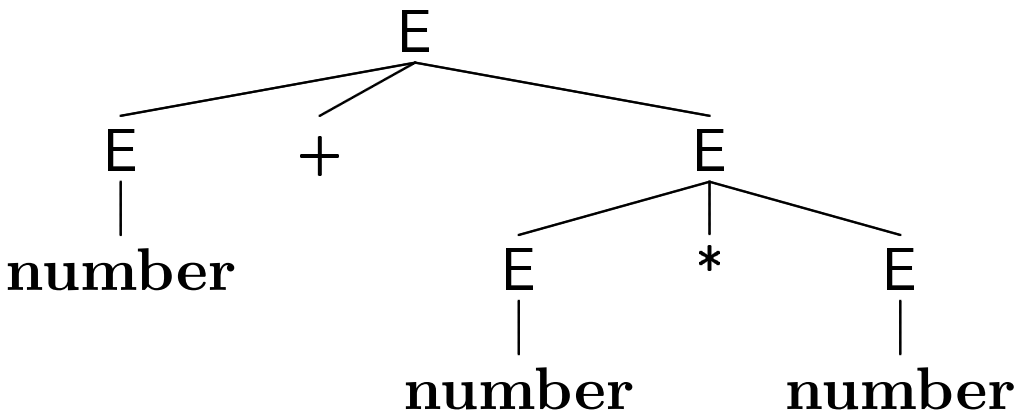
- Per comprendere questo codice (e, soprattutto, poterne scrivere altro!) è necessario riflettere ancora un poco proprio su produzioni e derivazioni.

Simboli e variabili interne

- In questa e nelle slide che seguono, quando parleremo di "simboli" della grammatica faremo esclusivo riferimento ai non terminali e ai simboli terminali dotati di valore semantico (numero e identificatori)
- Consideriamo dunque la derivazione canonica destra della stringa $\mathbf{number} + \mathbf{number} + \mathbf{number}$ nella grammatica della calcolatrice (tenendo presente le precedenze indicate)

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow E + E * E \\
&\Rightarrow E + E * \mathbf{number} \\
&\Rightarrow E + \mathbf{number} * \mathbf{number} \\
&\Rightarrow \mathbf{number} + \mathbf{number} * \mathbf{number}
\end{aligned}
$$

Simboli e istanze di simboli



- I differenti simboli utilizzati nella derivazione sono solo due, E e **number**
- Tuttavia, dal parse tree possiamo renderci conto che, se contiamo anche le "molteplicità", i simboli impiegati sono 8

Simboli e variabili interne

- Se distinguiamo usi distinti di uno stesso simbolo usando indici numerici, possiamo riscrivere i passaggi della derivazione nel modo seguente, in cui ad ogni passo mostriamo anche la produzione usata:

1.	$E_5 \Rightarrow E_1 + E_4$	$E_5 \to E_1 + E_4$
2.	$\Rightarrow E_1 + E_2 * E_3$	$E_4 \to E_2 * E_3$
3.	$\Rightarrow E_1 + E_2 * \mathbf{n}_3$	$E_3 \to \mathbf{n}_3$
4.	$\Rightarrow E_1 + \mathbf{n}_2 * \mathbf{n}_3$	$E_2 \to \mathbf{n}_2$
5.	$\Rightarrow \mathbf{n}_1 + \mathbf{n}_2 * \mathbf{n}_3$	$E_1 \to \mathbf{n}_1$

- La (apparentemente) strana numerazione dei simboli si comprende ricordando che il parser ricostruisce una derivazione canonica destra
- La logica per la numerazione segue semplicemente l'ordine delle riduzioni effettuate (colonna di destra, dal basso verso l'alto)
- Come si può vedere, ad ogni nuovo simbolo terminale incontrato ed ad ogni nuovo non terminale prodotto da una riduzione viene effettivamente assegnato un indice crescente

Simboli e variabili interne

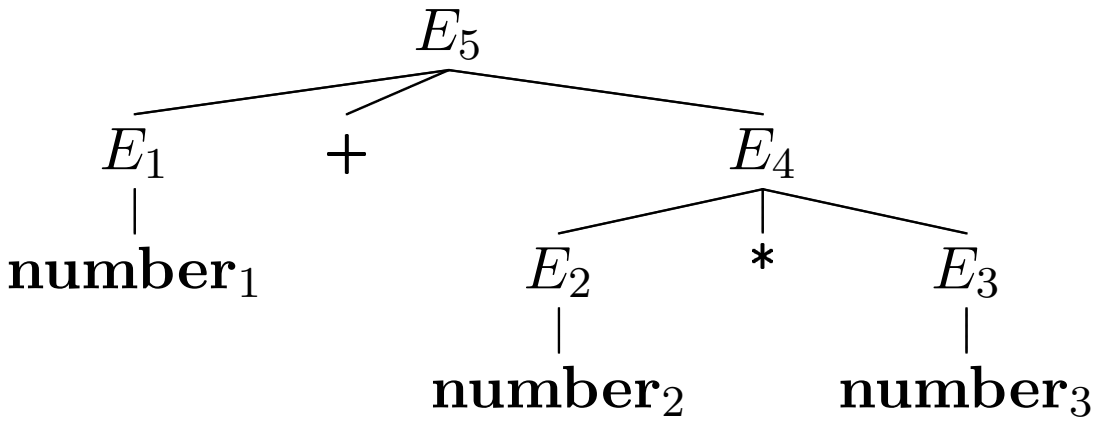
- Ad ogni nuovo simbolo Bison associa (almeno logicamente), una variabile interna (che, per semplicità, denoteremo allo stesso modo)
- Gli identificatori $$, $1, $2 che compaiono (o che possono comparire) nei frammenti di codice associati alle produzioni nei file Bison fanno proprio riferimento a tali variabili
- Ad esempio, nella regola (produzione + codice)

```
exp:  exp '+' exp  { $$ = $1 + $3; }
```

  i simboli $$, $1 e $3 denotano opportune variabili interne E_i, E_j e E_k, dove i valori precisi di i, j e k dipendono dalla sequenza di riduzioni che precedono l'uso della regola in questione
- In un successivo uso della stessa produzione le variabili denotate da $$, $1 e $3 saranno diverse (in tutto o in parte) da E_i, E_j e E_k
- Si noti che per individuare correttamente le variabili bisogna tenere presente che sono numerati anche i simboli privi di valore lessicale (dunque si dovrà usare $3 e non $2 per individuare la variabile associata all'ultimo non terminale exp)

Simboli e variabili interne

- Riconsideriamo il parse tree della sequenza $\mathbf{n} + \mathbf{n} * \mathbf{n}$, utilizzando questa volta gli indici per evidenziare il ruolo giocato dalle variabili interne



- Dopo la prima riduzione effettuata dal parser, e cioè

```
exp : 'number'
```

viene eseguita l'istruzione `$$ = $1` e questa, in termini delle variabili interne, equivale all'assegnamento $E_1 = \mathbf{number}_1$

- Se dunque la variabile $\mathbf{number}_1$ è inizializzata con il valore lessicale del corrispondente token, dopo la riduzione (e dunque dopo l'assegnamento), lo stesso valore è anche memorizzato in E_1

Simboli e variabili interne

- Analogamente, dopo le successive due riduzioni, che coinvolgono la stessa produzione

  ```
  exp : 'number'
  ```

  i valori lessicali (o semantici) dei token $\mathbf{number}_2$ e $\mathbf{number}_3$ verranno memorizzati nelle variabili E_2 ed E_3 perché il codice associato coinvolge differenti variabili.
- A questo punto dovrebbe essere chiaro che il codice eseguito dopo le due ulteriori riduzioni,

  ```
  exp : exp '*' exp      e      exp : exp '+' exp
  ```

  fa sì che il valore corretto dell'espressione venga memorizzato nella variabile E_5

L'istruzione di assegnamento

- La semplice calcolatrice che abbiamo realizzato consente di utilizzare variabili per memorizzare valori intermedi
- La produzione coinvolta è

  ```
  assignmente :'identifier' ':=' exp
  ```

  e il codice associato è `variables[$1]=$3`
- `variables` é un dizionario (map) definito nel main program e dunque il codice prevede proprio di memorizzare nel dizionario la coppia $< \texttt{identifier}_i, E_j >$, dove $\texttt{id}_i$ e E_j sono chiaramente le variabili associate a identificatore ed espressione "nel momento in cui il parser esegue la riduzione"
- Naturalmente poi il compilatore C++ utilizzerà i right value delle due variabili (il primo da usare come etichetta)

Interpreti vs compilatori

- La calcolatrice, realizzata nel modo appena visto, è un classico caso di interprete.
- Il programma "legge" la sequenza di istruzioni (espressioni e assegnamenti) e le esegue direttamente, senza produrre un codice oggetto
- Questi è in generale molto più semplice rispetto alla realizzazione di un compilatore completo (che produce codice eseguibile)
- Il parser però non produce direttamente il codice oggetto: esso produce un codice intermedio che poi viene fornito in input al middle-end
- Come prossimo step, nelle lezioni seguenti useremo Bison per realizzare un front-end che genera l'AST di programmi scritti nel linguaggio Kaleidoscope