Bison

Enea Zaffanella

enea.zaffanella@unipr.it

20 ottobre 2020

Linguaggi, interpreti e compilatori
Laurea Magistrale in Scienze informatiche

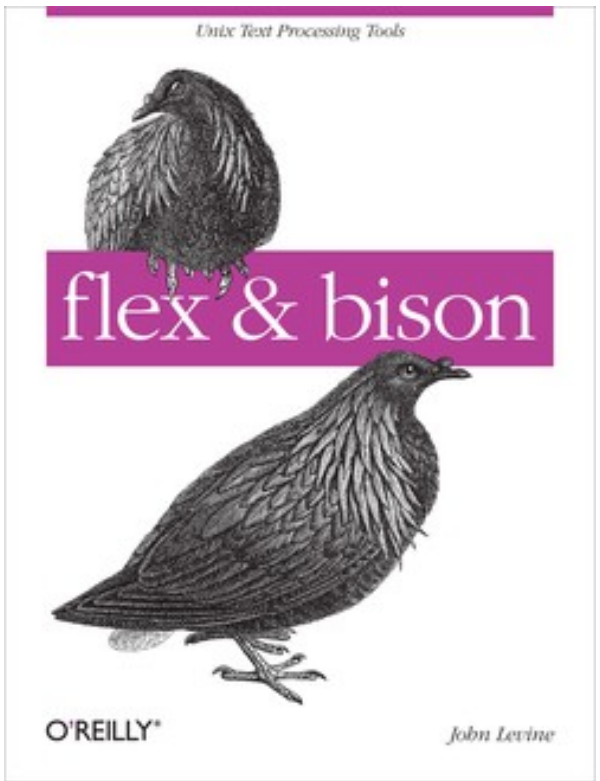
Sommario

Sommario

Lo strumento Bison

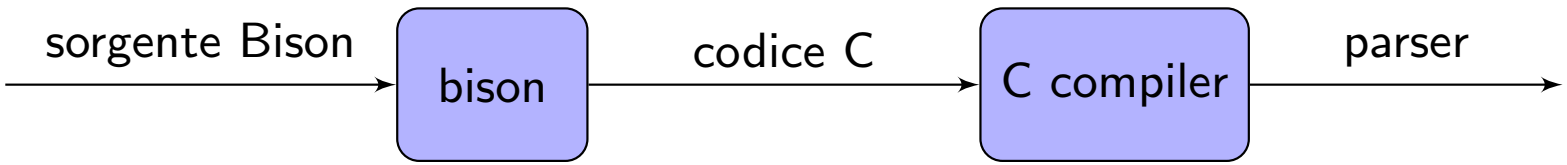
- Generatore di analizzatori sintattici (aka parser)
- Versione free di **Yacc** (1970)
- Produce codice C (oppure C++)
- Supporta l'uso combinato con **Flex**

RTFM: (Flex and) Bison



John Levine
flex & bison
O'Reilly, 2009

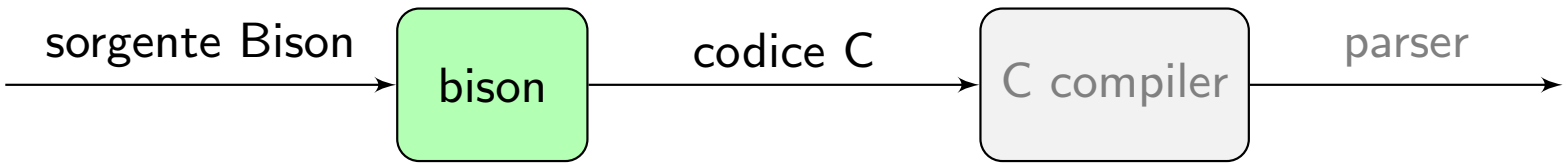
Generazione di un parser con Bison



Uso tipico

- `bison -o parser.c parser.yy`
- `gcc -Wall -Wextra -o parser parser.c`
- `parser < input_scanner`

Nota bene: anche Bison è un compilatore



Compilatore da L verso M

- codice sorgente L = linguaggio Bison
- codice "macchina" M = linguaggio C

Sommario

Il linguaggio sorgente Bison

Struttura del file sorgente: 3 sezioni

```
  /* Sezione delle definizioni */
%%
  /* Sezione delle regole */
%%
  /* Sezione del codice utente */
```

Sommario

La sezione definizioni

Cosa può contenere?

- "literal block"
- `%union` declaration
- `%start` declaration
- `%token` declaration (`%left`, `%right`, `%nonassoc`)
- `%type` declaration

Il literal block

Literal block

- un blocco di codice C racchiuso da %{ e %} (a inizio riga)
- viene copiato **verbatim** (i.e., letteralmente) nella parte iniziale del sorgente generato da Bison
- tipicamente può contenere:
  - inclusioni di file header o utente
  - dichiarazioni di variabili (usate nelle regole)
  - dichiarazioni di funzioni (invocate nelle regole)
  - definizione di funzioni **inline**

La union declaration

Sintassi

```
%union {
   tipo_1 nome_1;
   ...
   tipo_n nome_n;
};
```

Scopo

Definisce il tipo e nome degli attributi associati ai simboli (terminali e non-terminali) della grammatica

La start declaration

Sintassi

`%start` *non-terminale*

Scopo

Definisce lo *start symbol* della grammatica

Se non viene fornita

Si usa come *start symbol* il non-terminale nel membro sinistro della prima produzione elencata nella sezione delle regole

Le token declaration

Sintassi

`%token` [<*nome*>] *terminale*

oppure

`%token` [<*nome*>] *terminale* `"TOKEN"`

Scopo

Dichiara il simbolo come terminale e, se specificato, definisce il *nome* del membro della union che ne fornisce l'attributo; il lexer assegna il valore a `yylval.`*nome*

La seconda variante determina anche il lessema (`TOKEN`)

Esempio di token declaration

Nel file parser.yy

```
%union {
  double dval;
  /* ... */
};

%token <dval> NUMBER
```

Nel file lexer.ll

```
[0-9]+"."[0-9]*  {
                   yylval.dval = atof(yytext);
                   return NUMBER;
                 }
```

Le left, right, nonassoc declaration

Sintassi

`%left` [<*nome*>] *terminale*
`%right` [<*nome*>] *terminale*
`%nonassoc` [<*nome*>] *terminale*

Scopo

Dichiarano il simbolo come **terminale** e ne specificano l'associatività a sinistra, a destra o nulla; l'ordine (inverso) delle dichiarazioni ne stabilisce la precedenza

Le type declaration

Sintassi

`%type` <*nome*> *non-terminale*

Scopo

Dichiara il simbolo come non-terminale e definisce il *nome* del membro della union che ne fornisce l'attributo

Sommario

La sezione delle regole (i)

A cosa serve?

- Fornire la **definizione** della funzione `yyparse()`
- La funzione `int yyparse()` deve:
  - invocare l'analizzatore lessicale, chiamando `yylex()`, per ottenere i token (uno per ogni invocazione)
  - effettuare il parsing bottom up della sequenza di token
  - eseguire il codice associato ad ogni "riduzione" delle produzioni della grammatica effettuate durante in parsing
  - spesso, si calcola il valore dell'attributo per il lhs della produzione a partire da quelli calcolati per i simboli nel rhs

La sezione delle regole (ii)

Le regole sintattiche

- formato di una regola:

$$
\begin{array}{rll}
\mathit{nonterm}: & \mathit{rhs}_1 & \mathit{codice}_1 \\
| & \mathit{rhs}_2 & \mathit{codice}_2 \\
| & \ldots & \\
| & \mathit{rhs}_n & \mathit{codice}_n \\
; & &
\end{array}
$$

Esempio di regola

Una regola ricorsiva sinistra

```
expr :
    term
  | expr '+' term     { $$ = $1 + $3; }
  | expr '-' term     { $$ = $1 - $3; }
  ;
```

Note

- '+': indica un simbolo (terminale) letterale
- $$: indica il valore del simbolo nel lhs
- $*n*: indica il valore dell'*n*-esimo simbolo nel rhs
- regola di default (codice omesso): `$$ = $1`

Grammatiche accettate da Bison

- Normalmente, le grammatiche devono essere **non ambigue**
- Bison può accettare anche grammatiche ambigue, specificando **precedenza** e **associatività** degli operatori
  - **associatività**: `%left`, `%right`, `%noassoc`
  - **precedenza**: data dall'ordine (inverso) delle associatività
  - Esempio:

```
%left '+' '-'
%left '*' '/'
%right POW
```

- precedenza regole: quella del simbolo **terminale** più a **destra**

Sommario

La sezione del codice utente

- inizia dopo il secondo marker %%
- può contenere codice utente arbitrario, inserito **verbatim** dopo la definizione di `yyparse`
- tipicamente:
  - definizione delle funzioni ausiliarie precedentemente dichiarate (nella sezione delle definizioni)
  - la funzione `main` (non usuale)
- **best practice**: non mettere le definizioni delle funzioni, usare piuttosto un'altra unità di traduzione

Esempio sezione codice utente

```
  /* ... */
%%
  /* ... */
%%

int main() {
  return yyparse();
}
```

Opzioni a linea di comando

- `--report=all`: generazione report con indicazione della (eventuale) risoluzione dei conflitti
- `--defines=nome-header-file`: stabilisce il nome dell'header file generato (deve essere incluso nel file di input per Flex)
- `-o nome-file-output`: stabilisce il nome del file C/C++ generator da bison