Esame di

COMPILATORI ED INTERPRETI

A.A. 2007/08

REALIZZAZIONE DI UN PARSER PER PASCAL CON FLEX/BISON

Cortellino Marco, Gramegna Filippo, Loseto Giuseppe

Chiar.mo Prof. Giacomo Piscitelli

IL LINGUAGGIO PASCAL

L'applicazione proposta realizza un parser per il linguaggio ANSI Pascal utilizzando i generatori automatici: **FLEX** per l'analisi lessicale e **BISON** per l'analisi sintattica.

Il linguaggio **PASCAL** fu sviluppato sulla base dell'ALGOL 60 nei primi anni '70 da Niklaus Wirth. Comparato al suo predecessore il range di applicabilità è notevolmente migliorato grazie alla facilità di utilizzo delle strutture dati, caratteristica rara per i linguaggi dell'epoca. E' di fatti, utilizzato ancora oggi, specialmente in ambito didattico per la sua rigorosa e al contempo semplice struttura sintattica.

La struttura di un sorgente Pascal si compone di tre parti fondamentali:

1) **Intestazione del programma:** keyword **`program`** seguita dal nome
2) **Dichiarazioni:** è lo spazio in cui si dichiara tutto ciò che viene usato nel programma
3) **Istruzioni:** è lo spazio, delimitato dalle keyword **`begin`** ed **`end`** in cui si inseriscono le istruzioni del programma, ovvero il corrispettivo del main in C

Il punto più delicato è la definizione della parte delle dichiarazioni, dato che nel Pascal Standard esiste un ordine preciso nel tipo di istruzioni che possono essere inserite:

2.1) **Label** per utilizzo delle istruzioni **`goto`**
2.2) **Const** per la dichiarazione di costanti
2.3) **Type** per la dichiarazione dei tipi definiti dall'utente
2.4) **Var** per la dichiarazione di variabili
2.5) **Procedure** per la dichiarazione e definizione di procedure
2.6) **Function** per la dichiarazione e definizione di funzioni

Il Pascal prevede anche delle direttive al pre-processore, esse sono inserite sottoforma di commenti e dunque vengono completamente scartate nella fase di analisi sintattica.

Come quasi tutti i linguaggi di programmazione imperativi ad alto livello, anche il Pascal prevede le comuni istruzioni di:

1. **Loop**
   - **`while`** *condizione* **`do`** *istruzioni*
   - **`repeat`** *istruzioni* **`until`** *condizione*
   - **`for`** *ident* **`:=`** *expr* **`to`** *expr* **`do`** *istruzione*
2. **Diramazione**
   - **`if`** *condizione* **`then`** *istruzione*
   - **`if`** *condizione* **`then`** *istruzione* **`else`** *istruzione*
   - **`case`** *ident* **`of`** *expr1*`: istr1; …` *exprN*`: istrN;` **`end;`**
3. **Statement**
   - *istruzione;*
   - **`begin`** *istr1; … istrN;* **`end`**

ANALISI DELLA BNF

La BNF utilizzata è conforme all'ANSI Pascal ed è stata tratta dal manuale: "*Standard Pascal - User Reference Manual*", di Doug Cooper. Le estensioni rispetto a questa versione comprendono:

- Uso delle espressioni nella dichiarazione delle costanti
- Clausola **`otherwise`** nell'costrutto **`case`** (equivalente del **`default`** in C)
- Risoluzione dell'ambiguità nel costrutto **`if then else`** nella grammatica

In particolare relativamente a quest'ultimo punto, è noto che il costrutto **`if then else`** può generare ambiguità che può essere risolta associando per convenzione l'**`else`** all'ultimo **`if`** incontrato. Questo accorgimento può essere implementato in una fase successiva all'analisi sintattica con un approccio poco efficiente. La maniera corretta per gestire l'ambiguità invece è modificare la BNF in modo da rendere univoca l'interpretazione di questo tipo di costrutti.

In generale un costrutto if viene espresso nelle forme:

- **`if(`** `clause` **`)`** `statement`
- **`if(`** `clause` **`)`** `statement` **`else`** `statement`

- `statement:`
  - `simple_statement`
  - `if_statement`
  - `loop_statement`

Dove *simple statement* può essere un'espressione, un'istruzione composta, un'istruzione di *break* o *continue*, un loop *do/while*, o qualunque istruzione non ricorsiva (né a destra né a sinistra).

Loop statement può essere un *while statement* o un *for statement* in forma ricorsiva destra (o sinistra) come ad esempio:

- **`loop (`** `header` **`)`** `statement`

La sintassi illustrata è ambigua come mostrato dal seguente esempio

```
if (conditionA) if (conditionB) statementA; else statementB;
```

Stando alle regole scritte sopra, questa frase può essere interpretata sia come:

```
if (conditionA) {
  if (conditionB) statementA; else statementB;
}
```

Ma anche come:

```
if (conditionA) {
  if (conditionB) statementA;
}
else statementB;
```

Entrambe le interpretazioni sono sintatticamente corrette ma hanno significati molto differenti.

Per eliminare l'ambiguità nella grammatica possiamo riscrivere le regole nel modo seguente:

- statement
  - open_statement
  - closed_statement

- open_statement
  - **if(** clause **)** statement
  - **if(** clause **)** closed_statement **else** open_statement
  - **loop(** header **)** open_statement

- closed_statement
  - simple_statement
  - **if(** clause **)** closed_statement **else** closed_statement
  - **loop(** header **)** closed_statement

L'*open statement*, per definizione, contiene almeno un if non accoppiato con il relativo else. I closed statement invece, o non contengono affatto if, oppure contengono la forma dell'if accoppiata all'else. Con la sintassi così strutturata forziamo la presenza di soli *closed statement* tra le keyword *if* ed *else*, rendendo quindi sintatticamente corretta solo la prima forma dell'if vista in precedenza.

In altre parole tra un if ed un else è consentita la presenza dell'if solo se accopiato con un else. L'effetto finale è che ogni else è associato all'if che lo precede immediatamente.

Un caso particolare è rappresentato dal *loop statement* perché può essere *open* o *closed* a seconda delle istruzioni che lo seguono.

Un'altra comune sorgente di ambiguità nelle grammatiche è dovuta alla precedenza degli operatori, che può essere risolta più semplicemente definendo in maniera ordinata le regole di derivazioni delle espressioni all'interno della grammatica. Per ogni livello di precedenza utilizziamo un non terminale, rendendo le produzioni ricorsive a sinistra o a destra in funzione dell'associatività dell'operatore. L'idea è che se abbiamo un'espressione con un determinato livello di precedenza, non ci può essere una sottoespressione che contenga un operatore con un livello di precedenza più basso (a meno che questa non sia racchiusa tra parentesi tonde).

Ad esempio:

```
expression        : simple_expression | simple_expression relop simple_expression ;
simple_expression : term | simple_expression addop term ;
term              : factor | term mulop factor ;
factor            : sign factor | exponentiation;
exponentiation    : primary | primary STARSTAR exponentiation ;
primary           : identifier | constant | ( expression ) | NOT primary ;
```

IMPLEMENTAZIONE

Il parser realizzato sulla base di una grammatica BISON disponibile in [1]. L'analizzatore lessicale è stato generato utilizzando FLEX , che interagisce con BISON fornendogli iterativamente i token su richiesta di quest'ultimo, in particolare richiamando la funzione yylex() fino al termine dell'input.

Una particolarità dello scanner deriva dalla necessità di trattare gli elementi del linguaggio Pascal in maniera case-insensitive, così come richiesto dalla specifica. Questo requisito viene soddisfatto con delle definizioni del tipo:

```
A   [aA]
B   [bB]
…
Z   [zZ]
```

Oltre che il riconoscimento di keyword, identificatori, costanti ed operatori lo scanner si occupa anche di numerare le linee dell'input, e di eliminare i commenti.

I token vengono restituiti a Bison come delle costanti di tipo int, definite nel file y.tab.h incluso nel sorgente Flex con la direttiva:

```
%{ #include <y.tab.h> %}
```

Nella prima sezione del sorgente Bison sono definiti i token utilizzati dall'analizzatore lessicale, mediante la direttiva:

```
%token AND ARRAY ASSIGNMENT… …
```

Nella seconda parte è definita invece l'intera BNF della grammatica.

Attenzione particolare è stata rivolta alla gestione degli errori. In Bison è disponibile una funzione che viene richiamata in caso di errori: yyerror(char* s). Il parametro s contiene il tipo di errore che nella quasi totalità dei casi risulta essere un non meglio specificato "Syntax Error".

Per fornire maggiori informazioni sulle effettive cause di errori, è stato utilizzato un meccanismo di recovery degli errori reso disponibile dal simbolo speciale ***error*** inseribile all'interno di una regola.

Ad esempio:

```
program_heading :   PROGRAM identifier semicolon
                  | PROGRAM identifier LPAREN identifier_list RPAREN
                  | error identifier semicolon
                    {
                       vis_errore("Attesa la parola chiave PROGRAM\n");
                    }
                  | PROGRAM identifier error
                    {
                       vis_errore("';' atteso nell'intestazione del programma\n");
                    }
                  | PROGRAM error semicolon
                    {
                        vis_errore("Nome programma non valido\n");
                    };
```

Il simbolo error può sostituire uno o più simboli all'interno della regole. Qualora il simbolo sostituito non sia riconosciuto correttamente verranno eseguite le istruzioni indicate tra parentesi graffe alla fine della regola. Nel nostro caso, viene richiamata la funzione vis_errore, da noi definita nella prima sezione del sorgente Bison. Tale funzione ha il compito di visualizzare lo specifico errore ricevuto come parametro insieme al numero di riga in cui si è verificato e settare un apposito flag che indicherà la presenza di errori nel sorgente. E' stata definita anche la funzione yyerror, che viene richiamata automaticamente dal Bison in presenza di errori. Nel corpo di questa funzione viene semplicemente resettato a 0 il suddetto flag, tale operazione è necessaria al fine di ottenere la visualizzazione di ogni errore una sola volta. Infatti senza l'utilizzo di questo flag il messaggio di errore verrebbe segnalato ripetutamente dal punto in cui si è verificato fino alla radice del'albero sintattico.

Riferimenti:

[1] http://www.moorecad.com/standardpascal (Pascal.y e Pascal.lex)
[2] http://www.cs.stevens.edu/~badri/cs616/pas_bnf.html (BNF del pascal)