Parsing di "adding calculator" (ac): implementazione del parser a discesa ricorsiva

Paola Giannini

Riconoscimento stringhe del linguaggio

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

La grammatica di **ac**

0. $Prg \rightarrow DSs\ \$$
1. $DSs \rightarrow Dcl\ DSs$
2. $DSs \rightarrow Stm\ DSs$
3. $DSs \rightarrow \epsilon$
4. $Dcl \rightarrow Ty\ \text{id}\ DclP$
5. $DclP \rightarrow ;$
6. $DclP \rightarrow = Exp;$
7. $Stm \rightarrow \text{id} = Exp;$
8. $Stm \rightarrow \text{print id};$
9. $Exp \rightarrow Tr\ ExpP$
10. $ExpP \rightarrow -Tr\ ExpP$
11. $ExpP \rightarrow +Tr\ ExpP$
12. $ExpP \rightarrow \epsilon$
13. $Tr \rightarrow Val\ TrP$
14. $TrP \rightarrow /Val\ TrP$
15. $TrP \rightarrow *Val\ TrP$
16. $TrP \rightarrow \epsilon$
17. $Ty \rightarrow \text{float}$
18. $Ty \rightarrow \text{int}$

1X. $Val \rightarrow \text{intVal} \mid \text{floatVal} \mid \text{id}$

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

La tabella Predict

Il lessico

Token	Simboli usati nella grammatica
INT	intVal
FLOAT	floatVal
ID	id
TYINT	int
TYFLOAT	float
ASSIGN	=
PRINT	print
PLUS	+
MINUS	-
TIMES	*
DIVIDE	/
SEMICOLON	;
EOF	$

Num.	LHS	RHS	Predict
0.	*Prg*	*DSs* $	{float,int,id,print,$}
1.	*DSs*	*Dcl DSs*	{float,int}
2.	*DSs*	*Stm DSs*	{id,print}
3.	*DSs*	ϵ	{$}
4.	*Dcl*	*Ty* id*DclP*	{float,int}
5.	*DclP*	;	{;}
6.	*DclP*	= *Exp*;	{=}
7.	*Stm*	id = *Exp*;	{id}
8.	*Stm*	print id;	{print}
9.	*Exp*	*Tr ExpP*	{intVal,floatVal,id}
10.	*ExpP*	+*Tr ExpP*	{+}
11.	*ExpP*	-*Tr ExpP*	{-}
12.	*ExpP*	ϵ	{;}
13.	*Tr*	*Val TrP*	{intVal,floatVal,;}
14.	*TrP*	**Val TrP*	{*}
15.	*TrP*	/*Val TrP*	{/}
16.	*TrP*	ϵ	{+,-,;}
17.	*Ty*	float	{float}
18.	*Ty*	int	{int}
19.	*Val*	intVal	{intVal}
20.	*Val*	floatVal	{floatVal}
21.	*Val*	id	{id}

Aggiunte alla classe Scanner

- Alla classe Scanner dobbiamo aggiungere il metodo peekToken() che restituisce il prossimo token, ma non consuma l'input, in modo tale che una successiva nextToken() o peekToken() restituisca lo stesso token.
- Come lo definiamo?
- Testate la corretta esecuzione della peekToken(), cioè che non consuma l'input. Ad esempio:

```
@Test
void peekToken () {
  Scanner s = new Scanner ("......./testGenerale.txt");
  assertEquals(s.peekToken().getType(), TokenType.TYINT );
  assertEquals(s.nextToken().getType(), TokenType.TYINT );
  assertEquals(s.peekToken().getType(), TokenType.ID );
  assertEquals(s.peekToken().getType(), TokenType.ID );
  Token t = s.nextToken();
  assertEquals(t.getType(), TokenType.ID);
  assertEquals(t.getRiga(), 1);
  assertEquals(t.getVal(), "temp");
}
```

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

Parser

Definiamo in un package `parser`

- la classe `Parser` che avrà
  - un costruttore che prende come input uno `Scanner` da memorizzare in un campo privato, `scanner`,
  - i metodi `parseNT` per ogni nonterminale della grammatica che restituiscono **`void`** se non ci sono stati errori oppure segnalano un errore che deve dire "quale è il token che causa l'errore e perchè"
  - il metodo `match` descritto a lezione
  - il metodo `parse` che ritorna `parsePrg`

Pseudocodice di `match` e `parsePrg`

`match` deve controllare se il prossimo token ha un certo tipo nel qual caso lo consuma e lo ritorna altrimenti da errore.

ATTENZIONE: Questo è pseudocodice, per cui non c'è trattamento delle eccezioni!

```
Token match(TokenType type) {
  Token tk = peekToken();
  if (type.equals(tk.getType())) return nextToken;
  else ErroreSintattico:
    // aspettato "type" token invece di tk alla riga tk.getRiga()
}
```

`parsePrg` ritorna senza dare errori se il programma è sintatticamente corretto

```
void parsePrg(){
  Token tk=peekToken()
  switch (tk.getType()) {
    case TokenType.TYFLOAT:
    case TokenType.TYINT:
    case TokenType.ID:
    case TokenType.PRINT:
    case TokenType.EOF:
      parseDSs();
      match(TokenType.EOF);
      return;
   }
 ErroreSintattico:
    // token tk alla riga tk.getRiga() non e' l'inizio di un programma
}
```

UPO UNIVERSITÀ DEL PIEMONTE ORIENTALE

IMPORTANTE: Definizione incrementale

Iniziate definendo i seguenti metodi

1. `parsePrg`
2. `parseDSs`
3. `parseDcl`
4. `parseDclP` considerando SOLO la produzione 5: $DclP \rightarrow$;
5. `parseStm` considerando SOLO la produzione 8: $Stm \rightarrow$ print id;

SOLAMENTE quando questi funzionano correttamente aggiungete il parsing della dichiarazione con inizializzazione (cioè la produzione 6) e dell'assegnamento (cioè la produzione 7) e i metodi per gli altri non terminali

5. `parseExp`
6. `parseExpP`
7. `parseTr`
8. `parseTrP`
9. `parseVal`

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

Test

Aggiungiamo al package `test`

- una classe di test `TestParser` che testa
  - il parsing, costruendo uno Scanner su files che siano corretti dal punto di vista lessicale, ma alcuni corretti dal punto di vista sintattico altri no
  - per il momento potete testare che su programmi sintatticamente corretti il parser non lanci eccezioni (oppure abbia una stringa di errore vuota!) e che invece su programmi sintatticamente scorretti lanci la giusta eccezione (oppure ritorni la giusta stringa di log)

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE