TUTORIAL SU COME FARE LE FACCE

Trovare i NULLABLE:

- Bisogna controllare se da una produzione è possibile derivare solo ε.

Trovare i FIRST:

- Per ogni non-terminale in ogni produzione, se trovo un non-terminale
  - NULLABLE, aggiungo i suoi FIRST, e vado al prossimo simbolo.
  - non-NULLABLE, aggiungo i suoi FIRST, e mi fermo.

  se il prossimo simbolo è un terminale, lo aggiungo e mi fermo.

Trovare i FOLLOW:

- Aggiungere la produzione iniziale del tipo: A' → A$
- Per ogni non-terminale in ogni produzione, prendo i FIRST del resto della produzione: se il resto della produzione è anche NULLABLE, aggiungo ai FOLLOW del mio non-terminale i FOLLOW del non-terminale di partenza che genera la produzione corrente.

Risolvere la left-recursion:

- Aggiungo un nuovo simbolo, e sostituisco uno dei vecchi simboli con il nuovo, e metto il nuovo in fondo:

$E \rightarrow E\ E\ +$ $E \rightarrow E\ E\ *$ $E \rightarrow num$	$E \rightarrow num\ E_1$ $E_1 \rightarrow E\ +\ E_1$ $E_1 \rightarrow E\ *\ E_1$ $E_1 \rightarrow \varepsilon$

Fattorizzazione a sinistra:

- Sostituire i non-terminali che si ripetono con nuovo simbolo non-terminale:

$E \rightarrow num\ E_1$ $E_1 \rightarrow E\ +\ E_1$ $E_1 \rightarrow E\ *\ E_1$ $E_1 \rightarrow \varepsilon$	$E \rightarrow num\ E_1$ $E_1 \rightarrow E\ AUX$ $AUX \rightarrow +\ E_1$ $AUX \rightarrow *\ E_1$ $E_1 \rightarrow \varepsilon$

Trovare i look-ahead:

- Sono i FIRST della produzione; se questa è NULLABLE, lo sono anche i FOLLOW del non terminale a sinistra.

Risolvere gli SLR:

- Aggiungere la produzione iniziale del tipo: A' → A$.
- Costruire FIRST E FOLLOW.
- Disegnare l'NFA di ogni produzione.
- Scrivere la tabella delle ε-closures, a partire dal grafico, collegando tutti gli stati da cui esce un non terminale a tutti gli stati iniziali delle produzioni con un non-TERMINALE a sx uguale al non terminale cercato.
- Costruire il DFA, ovvero:
  - Fare la ε-closure della produzione iniziale.
  - Fare le move a partire dalla ε-closure della produzione iniziale, con tutti i terminali e non terminali di tutte le produzioni.
- Costruire la tabella del parser, dove si hanno gli stati e le rispettive ε-closures sulla colonna, e sulla riga tutti i simboli terminali e non:
  - Si mettono gli shift (S1,S2,...) sui simboli terminali, i goto (G1,G2, ...) sui non terminali.
  - ACCEPT (ACC) sul $ dove è contenuto lo stato terminale dell'NFA solitamente B (produzione iniziale).
  - REDUCE (r1, r2, ...) degli stati su un simbolo presente nei FOLLOW della produzione, col numero dell'NFA in cui è contenuto lo stato finale.

Esempio: data la produzione, costruire la tabella di parsing SLR

T→T⇒T T→T*T T→int

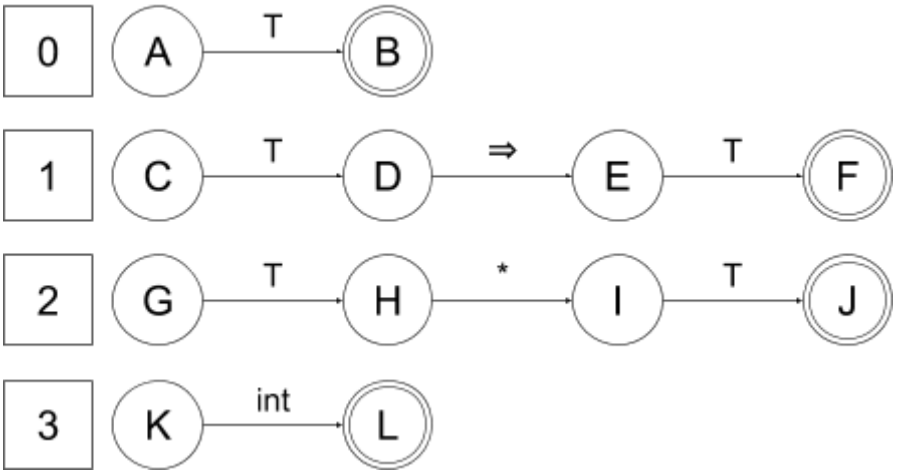
1. Numerare le produzioni e aggiungere la produzione iniziale

T'→T$	0
T→T⇒T	1
T→T*T	2
T→int	3

2. Calcolare FIRST e FOLLOW

First(T)={int}	Follow(T)={⇒, *}

3. Costruire gli NFA



4. Collegare gli NFA: scrivere la tabella delle ε-closures, a partire dal grafico, collegando tutti gli stati da cui esce un non terminale a tutti gli stati iniziali delle produzioni con un non-TERMINALE a sx uguale al non terminale cercato

STATO	**ε**
A	CGK
C	CGK
G	CGK
E	CGK
I	CGK

5. Costruire la tabella delle move: mettere ACC nell'NFA con lo stato finale della produzione $

#DFA	**NFA**	**int**	*****	**⇒**	**T**	**$**
0	A C G K	s1			g2	
1	L					
2	B D H		s3	s4		ACC
3	I C G K	s1			g5	
4	E C G K	s1				
5	J D H		s3	s4		
6	F D H		s3	s4		

6. Scrivere le reduce: negli NFA che hanno uno stato finale, nelle caselle dei follow, mettere reduce con il numero dell'NFA in cui è contenuto lo stato finale.

#DFA	NFA	int	*	⇒	T	$
0	A C G K	s1			g2	
1	L		r3	r3		r3
2	B D H		s3	s4		ACC
3	I C G K	s1			g5	
4	E C G K	s1				
5	J D H		r2/s3	r2/s4		r2
6	F D H		r1/s3	r1/s4		r1

7.

Tabella Espressioni

$$
\begin{array}{lll}
S \rightarrow \mathbf{id} = E\ ; & \{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \} \\
\quad \mid\ L = E\ ; & \{\ gen(L.array.base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \} \\
E \rightarrow E_1 + E_2 & \{\ E.addr = \mathbf{new}\ Temp\,();\ gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
\quad \mid\ \mathbf{id} & \{\ E.addr = top.get(\mathbf{id}.lexeme);\ \} \\
\quad \mid\ L & \{\ E.addr = \mathbf{new}\ Temp\,();\ gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \} \\
L \rightarrow \mathbf{id}\ [\ E\ ] & \{\ L.array = top.get(\mathbf{id}.lexeme);\ L.type = L.array.type.elem;\ L.addr = \mathbf{new}\ Temp\,();\ gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \} \\
\quad \mid\ L_1\ [\ E\ ] & \{\ L.array = L_1.array;\ L.type = L_1.type.elem;\ t = \mathbf{new}\ Temp\,();\ L.addr = \mathbf{new}\ Temp\,();\ gen(t\ '='\ E.addr\ '*'\ L.type.width);\ \} \\
 & \quad gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}
\end{array}
$$

Tabella Booleani

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \ \texttt{||} \ B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \ \|\| \ label(B_1.false) \ \|\| \ B_2.code$
$B \rightarrow B_1 \ \&\& \ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \ \|\| \ label(B_1.true) \ \|\| \ B_2.code$
$B \rightarrow \texttt{!} \ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \ \textbf{rel} \ E_2$	$B.code = E_1.code \ \|\| \ E_2.code$ $\|\| \ gen('\texttt{if}' \ E_1.addr \ \textbf{rel}.op \ E_2.addr \ '\texttt{goto}' \ B.true)$ $\|\| \ gen('\texttt{goto}' \ B.false)$
$B \rightarrow \textbf{true}$	$B.code = gen('\texttt{goto}' \ B.true)$
$B \rightarrow \textbf{false}$	$B.code = gen('\texttt{goto}' \ B.false)$

Tabella Costrutti

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$
$S \rightarrow$ **while** (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

Type checker

```
In.Iota (n_exp, pos)
     => let val (e_type, n_exp_dec) = checkExp ftab vtab n_exp
        in if e_type = Int
           then (Array Int, Out.Iota (n_exp_dec, pos))
           else raise Error ("Iota: wrong argument type " ^
                             ppType e_type, pos)
        end
```

Interprete

```
evalExp ( Iota (e, pos), vtab, ftab ) =
       let val sz = evalExp(e, vtab, ftab)
       in case sz of
            IntVal size =>
              if size >= 0
              then ArrayVal(List.tabulate(size, (fn x => IntVal x)),
                            Int)
              else raise Error("Error: In iota call, size is negative: "
                               ^ Int.toString(size), pos)
          | _ => raise Error("Iota argument is not a number: "^ppVal 0 sz, pos)
       end
```

```
| Exp PLUS  Exp  { Plus (#1 $1, $3, #2 $1) }
| Exp MINUS Exp  { Minus(#1 $1, $3, #2 $1) }
| Exp TIMES Exp  { Times  (#1 $1, $3, #2 $1) }
| Exp DIV Exp    { Divide (#1 $1, $3, #2 $1) }
| Exp AND Exp    { And (#1 $1, $3, #2 $1) }
| Exp OR Exp     { Or (#1 $1, $3, #2 $1) }
| Exp DEQ   Exp  { Equal(#1 $1, $3, #2 $1) }
| Exp LTH   Exp  { Less (#1 $1, $3, #2 $1) }
| NOT Exp        { Not (#2 $1, #1 $1) }
| NEGATE Exp     { Negate (#2 $1, #1 $1) }
| IF Exp THEN Exp ELSE Exp %prec ifprec
                 { If (#2 $1, $4, $6, #1 $1) }
| ID LPAR Exps RPAR
                { Apply (#1 $1, $3, #2 $1) }
| ID LPAR RPAR  { Apply (#1 $1, [], #2 $1) }
| READ LPAR Type RPAR
                { Read (#3 $1, #1 $1) }
| WRITE LPAR Exp RPAR
                { Write (#3 $1, (), #1 $1) }
| IOTA LPAR Exp RPAR
                { Iota (#3 $1, #1 $1) }
| REDUCE LPAR FunArg COMMA Exp COMMA Exp RPAR
                { Reduce (#3 $1, $5, $7, (), #1 $1) }
| MAP LPAR FunArg COMMA Exp RPAR
                { Map (#3 $1, $5, (), (), #1 $1) }
| REDUCE LPAR OP BinOp COMMA Exp COMMA Exp RPAR
                { Reduce (#4 $1, $6, $8, (), #1 $1) }
| MAP LPAR UnOp COMMA Exp RPAR
                { Map (#3 $1, $5, (), (), #1 $1) }
| LPAR Exp RPAR { $2 }
| LET ID EQ Exp IN Exp %prec letprec
                { Let (Dec (#1 $2, $4, $3), $6, #1 $1) }

--------------------------------
In.Iota (n_exp, pos)
     => let val (e_type, n_exp_dec) = checkExp ftab vtab n_exp
        in if e_type = Int
           then (Array Int, Out.Iota (n_exp_dec, pos))
           else raise Error ("Iota: wrong argument type " ^
                             ppType e_type, pos)
        end
```

```
| In.Plus (e1, e2, pos)
     => let val (_, e1_dec, e2_dec) = checkBinOp ftab vtab (pos, Int, e1, e2)
        in (Int,
            Out.Plus (e1_dec, e2_dec, pos))
        end

| In.If (pred, e1, e2, pos)
     => let val (pred_t, pred') = checkExp ftab vtab pred
            val (t1, e1') = checkExp ftab vtab e1
            val (t2, e2') = checkExp ftab vtab e2
            val target_type = checkTypesEqualOrError pos (t1, t2)
        in case pred_t of
               Bool => (target_type,
                        Out.If (pred', e1', e2', pos))
             | otherwise => raise Error ("Non-boolean predicate", pos)
        end
---------------------------
evalExp ( Iota (e, pos), vtab, ftab ) =
       let val sz = evalExp(e, vtab, ftab)
       in case sz of
            IntVal size =>
               if size >= 0
               then ArrayVal(List.tabulate(size, (fn x => IntVal x)),
                             Int)
               else raise Error("Error: In iota call, size is negative: "
                                ^ Int.toString(size), pos)
          | _ => raise Error("Iota argument is not a number: "^ppVal 0 sz, pos)
       end

| evalExp ( Plus(e1, e2, pos), vtab, ftab ) =
       let val res1   = evalExp(e1, vtab, ftab)
           val res2   = evalExp(e2, vtab, ftab)
       in  case (res1, res2) of
             (IntVal n1, IntVal n2) => IntVal (n1+n2)
           | _ => invalidOperands "Plus on non-integral args: " [(Int, Int)] res1 res2 pos
       end

| evalExp ( If(e1, e2, e3, pos), vtab, ftab ) =
       let val cond = evalExp(e1, vtab, ftab)
       in case cond of
             BoolVal true  => evalExp(e2, vtab, ftab)
          |  BoolVal false => evalExp(e3, vtab, ftab)
          |  other         => raise Error("If condition is not a boolean", pos)
       end
```