Laboratorio 2 - SmallC: Esecutore OCaml
Sintassi Astratta e Concreta: Scrittura e Stampa dei Prog.

Sommario: 29 Marzo, 2019

- Sviluppo in OCaml
- Sintassi Astratta: Scrittura e Stampa di Programmi SmallC mediante Generazione dei loro AST.
- Sintassi Concreta: Stampa di AST in sintassi concreta SmallC.
- Attività di Oggi: 3 Esercizi

Sviluppo in OCaml.
Operiamo a Top Level Evaluation (Interpretation Cycle)

- Eseguiamo ocaml in una shell Unix ed entriamo in Top Level Evaluation

- Ambiente Interattivo: Un blocco virtuale, Scope Statico, Un ciclo:
  - Input: # Termine T (i.e. Espressione, Programma) seguito da ;;
  - Valutazione di T
    - T è analizzato dal FrontEnd OCaml
    - T è Compilato in Bytecode OCaml, ottenendo T_B
    - T_B è Interpretato e valutato ad un valore V
  - output: *nome*: *tipo* = V

- Il ciclo input-valutazione-output si ripete:
  - Fintantochè T è OCaml-corretto e valutabile ad un valore;
  - Valutazioni Non Terminanti per termini divergenti;
  - control-c per interrompere esecuzione e tornare alla fase input del ciclo;
  - exit(n);; – con n intero per chiudere il ciclo ed uscire dall'Ambiente.

Programmare in OCaml

- OCaml è un Linguaggio Multiparadigma:
  - sottoLinguaggio Funzionale (puro): Calcolo Funzionale
  - sottoLinguaggio Imperativo: stato, modificabili e sequenza;
  - sottoLinguaggio Object Oriented: Classi, Oggetti ed Ereditarietà;
  - integrati in accordo a specifiche regole di composizione dei costrutti

- Nel Laboratorio useremo solo il sottoLinguaggio Funzionale

- Abbiamo visto (in Laboratorio1 del 22/03, e a Lezione-Esercitazione del 26/03)
  - Operatori primitivi e binari, infissi usabili anche in forma prefissa.
  - Naming per valori costanti: Costrutto "let ide = T".
  - Valori Primitivi Atomici per tipi: int, bool, char.
  - Linguaggio fortemente tipato: Ogni termine ha uno e un solo Tipo.
  - Il Tipo determina le operazioni applicabili ad un valore e l'uso ammesso.
  - Valori Primitivi Strutturati per tipi: string, tuple e liste.
  - Valori Funzione: Astrazioni fun, Naming, Monadiche e Curried.
  - Ricorsione e operatore rec
  - Tipi Algebrici o Concreti: Costruttori di Valori
  - Costrutto match per accesso, visita e "modifica" di valori
  - Valori Funzione: Definizione per casi e Costrutto function.
  - Cautela nella sintassi concreta e Convenzioni sull'uso degli identificatori.

OCaml: Uso nel Laboratorio

- **OCamlIde.**
  (a) Integraded Development Environment per OCaml: Ne esistono diversi.
  (b) Offrono un sistema di Editing-Esecuzione-Mofifica-Versionamento
  (c) Non li useremo: Li suppliremo in modo (da) manuale.

- **SmallC.ml.**
  (a) file residente sulla directory di lavoro, **WAD**, per tutta l'intera attività di Sviluppo;
  (b) distribuito con un Contenuto Iniziale, e da caricare oggi, sulla propria WAD;

- **Ad ogni sessione di lavoro.**
  (a) SmallC.ml è copiato e la copia rinominata in SmallCxxx.ml, dove xxx è un progressivo a partire da 0;
  (b) Ocaml è avviato da WAD;
  (c) Caricamento di SmallC.ml: `# #use "SmallC.ml";;`

- **Editing del Codice.**
  - TLE legge ed analizza carattere per carattere editato dopo il prompt #
  - Nessuna possibilità di correzione in linea;
    (a) tenera aperta una finestra di editing su SmallC.ml;
    (b) editare il proprio codice su SmallC.ml;
    (c) per l'interpretazione di quanto editato procedere con 1 o con 2:
      (1) copy and past del codice editato;
      (2) ri-caricamento del file.

Sintassi Astratta: Una grammatica per alberi AST

$$\texttt{Dcl} ::= [\texttt{const}]\ \texttt{Ide}\ \texttt{Num} \mid [\texttt{var}]\ \texttt{Ide}\ \texttt{Num} \mid [\texttt{varN}]\ \texttt{Ide} \mid [\texttt{array}]\ \texttt{Ide}\ \texttt{Num} \mid [\texttt{emptyDCL}] \mid \texttt{Dcl}\ [\texttt{seqD}]\ \texttt{Dcl}$$

$$\begin{aligned}\texttt{Exp} ::= {} & [\texttt{val}]\ \texttt{Ide} \mid \texttt{Num} \mid \texttt{Ide}\ [\uparrow]\ \texttt{Exp} \\ & \mid \texttt{Exp}\ [+]\ \texttt{Exp} \mid \texttt{Exp}\ [-]\ \texttt{Exp} \mid \texttt{Exp}\ [*]\ \texttt{Exp} \mid \texttt{Exp}\ [\texttt{div}]\ \texttt{Exp} \\ & \mid \texttt{Exp}\ [==]\ \texttt{Exp} \mid \texttt{Exp}\ [<]\ \texttt{Exp} \mid \texttt{Exp}\ [>]\ \texttt{Exp} \\ & \mid [\texttt{not}]\ \texttt{Exp} \mid \texttt{Exp}\ [\texttt{or}]\ \texttt{Exp} \mid \texttt{Exp}\ [\texttt{and}]\ \texttt{Exp}\end{aligned}$$

$$\begin{aligned}\texttt{Cmd} ::= {} & [\texttt{ifE}]\ \texttt{Exp}\ \texttt{Cmd}\ \texttt{Cmd} \mid [\texttt{ifT}]\ \texttt{Exp}\ \texttt{Cmd} \\ & \mid \texttt{Ide}\ [=]\ \texttt{Exp} \mid \texttt{Ide}\ \texttt{Exp}\ [\leftarrow]\ \texttt{Exp} \\ & \mid [\texttt{while}]\ \texttt{Exp}\ \texttt{Cmd} \mid \texttt{Cmd}\ [\texttt{seqC}]\ \texttt{Cmd} \mid [\texttt{emptyCMD}]\end{aligned}$$

$$\texttt{Prog} ::= [\texttt{prog}]\ \texttt{Dcl}\ \texttt{Cmd} \mid [\texttt{progN}]\ \texttt{Cmd}$$

where:
– [xxx] indica indica un costruttore di alberi AST di nome xxx;
– $\uparrow$ costruttore termine (AST) accesso valore componente array;
– $\leftarrow$ costruttore termine modifica valore componente array;

Sintassi Concreta di SMallC: Una Grammatica per Sequenze di Tokens

$$\texttt{Dcl} ::= \texttt{ide} = \texttt{num} \mid \texttt{var ide} = \texttt{num} \mid \texttt{var ide} \mid \texttt{ide[num]}$$
$$\texttt{Dcls} ::= \epsilon \mid \texttt{Dcl;Dcls}$$

$$\texttt{ExpB2} ::= \texttt{ExpB2 or ExpB1} \mid \texttt{ExpB1}$$
$$\texttt{ExpB1} ::= \texttt{ExpB1 and ExpB} \mid \texttt{ExpB}$$
$$\texttt{ExpB} ::= \texttt{not ExpB} \mid \texttt{ExpR}$$
$$\texttt{ExpR} ::= \texttt{ExpR} == \texttt{ExpA2} \mid \texttt{ExpR} < \texttt{ExpA2} \mid \texttt{ExpR} > \texttt{ExpA2} \mid \texttt{ExpA2}$$
$$\texttt{ExpA2} ::= \texttt{ExpA2} + \texttt{ExpA1} \mid \texttt{ExpA2} - \texttt{ExpA1} \mid \texttt{ExpA1}$$
$$\texttt{ExpA1} ::= \texttt{ExpA1} * \texttt{ExpA} \mid \texttt{ExpA1 div ExpA} \mid \texttt{ExpA}$$
$$\texttt{ExpA} ::= \texttt{ide} \mid \texttt{num} \mid \texttt{ide[ExpA2]} \mid \texttt{(ExpB2)}$$

$$\texttt{Cmd} ::= \texttt{if (ExpB2) Cmd; else Cmd;} \mid \texttt{OtherCmd;}$$
$$\texttt{OtherCmd} ::= \texttt{if (ExpB2) OtherCmd} \mid \texttt{NonCondionalCmd}$$
$$\texttt{NonCondionalCmd} ::= \texttt{ide} = \texttt{ExpA2} \mid \texttt{Ide[ExpA2]} = \texttt{ExpA2}$$
$$\mid \texttt{while (ExpB2)} \ \widehat{\{}\ \texttt{Cmd}\ \widehat{\}} \mid \{\texttt{Cmd}\}$$
$$\texttt{Cmds} ::= \texttt{Cmd} \mid \texttt{Cmd Cmds}$$

$$\texttt{Prog} ::= \widehat{\{}\ \texttt{Dcls}\ \{\texttt{Cmd}\}\ \widehat{\}}$$

where: `ide,=,num,var,[, ],;,or,and,not,==,<,>,+,-,*,div,(,),if,then,else,while,`$\widehat{\{}$`,`$\widehat{\}}$ sono categorie lessicali (contenenti 1 solo lessema, ad eccezione di ide e num): $\widehat{\{}$ ha lessema "{", $\widehat{\}}$ ha lessema "}".

Sintassi Concreta: Il Programma P0 di SMallC

- Il programma P0 scritto nella Sintassi Concreta della Grammatica di SMallC scritta sopra in EFNF.
- Indentazioni, Spazi, e Nuova Linea possono essere arbitrariamente scelte (dallo autore) come nel Nuova Linea prima del simbolo "else" .

```
{
 x = 7;
 var y = 10;
 var z;
 A[12];
 y = x+y;
 if (x+y < 15) z = y;
 else z = 0;
 while z<12 {
        A[z] = y+z;
        z = z+1
        }
 }
```

Front-End di SMallC: Scrittura del Programma P0

- Il programma P0 è scritto direttamente utilizzando le funzioni per la costruzione di AST del Front-End dell'interprete SMallC.
- In figura una sessione OCaml in cui usiamo le funzioni del Front-End SMallC prima per scrivere P0 in Sintassi Astratta e poi, per ottenere l'AST di P0.

```
# let d1 = Const("x",7) in
let d2 = Var("y",10) in
let d3 = VarN "z" in
let d4 = Array("A",12) in
let dd = SeqDcl(d1,SeqDcl(d2,SeqDcl(d3,d4))) in
let c1 = UpdVar("y",Plus(Val "x",Val "y")) in
let e21 = LT (Plus(Val "x",Val "y"),Num 15) in
let c22 = UpdVar("z",Val "y") in
let c23 = UpdVar("z",Num 0) in
let c2 = IfE(e21,c22,c23) in
let e31 = LT (Val "z", Num 12) in
let c321 = UpArray("A",Val "z",Plus(Val "y",Val "z")) in
let c322 = UpdVar("z",Plus(Val "z",Num 1)) in
let c32 = SeqCmd(c321,c322) in
let c3 = While(e31,c32) in
let cc = SeqCmd(c1,SeqCmd(c2,c3)) in
let p0 = Program(dd,cc) in
p0;;
- : prog =
Program
 (SeqDcl (Const ("x", 7),
   SeqDcl (Var ("y", 10), SeqDcl (VarN "z", Array ("A", 12)))),
 SeqCmd (UpdVar ("y", Plus (Val "x", Val "y")),
  SeqCmd
   (IfE (LT (Plus (Val "x", Val "y"), Num 15), UpdVar ("z", Val "y"),
     UpdVar ("z", Num 0)),
   While (LT (Val "z", Num 12),
    SeqCmd (UpArray ("A", Val "z", Plus (Val "y", Val "z")),
     UpdVar ("z", Plus (Val "z", Num 1)))))))
```

SMallC: Stampa del Programma P0

- Il programma P0 è scritto direttamente utilizzando le funzioni per la costruzione di AST del Front-End dell'interprete SMallC.
- printProg converte lo AST di P0 in una presentazione di P0 in Sintassi Astratta.

```
p0;;
- : prog =
Program
 (SeqDcl (Const ("x", 7),
   SeqDcl (Var ("y", 10), SeqDcl (VarN "z", Array ("A", 12)))),
 SeqCmd (UpdVar ("y", Plus (Val "x", Val "y")),
  SeqCmd
   (IfE (LT (Plus (Val "x", Val "y"), Num 15), UpdVar ("z", Val "y"),
     UpdVar ("z", Num 0)),
   While (LT (Val "z", Num 12),
    SeqCmd (UpArray ("A", Val "z", Plus (Val "y", Val "z")),
     UpdVar ("z", Plus (Val "z", Num 1)))))))
# printProg p0;;

{
 x = 7;
 var y = 10;
 var z;
 A[12];
 y = (x + y);
 if ((x + y) < 15) {
      z = y;
      }
 else  z = 0;
 while (z < 12) {
        A[z] = (y + z);
        z = (z + 1);
        }
}
```

SmallC: 3 Esercizi

Esercizio (1)

Scrivere nello spazio riservato ai "Tests: Abstract Syntax" le definizioni che si ritengano appropriate per definire l'AST del programma Uno di SmallC:

```
{ var p1 = 150;
  var mis = 30;
  lun = 7;
  p1 = lun + mis * p1;
}
```

Si mostri, quindi l'AST costruito.

Esercizio (2)

Applicare printProg all'AST ottenuto in esercizio1. Commentare la stampa ottenuta e correggere la definizione di toStringDcl in modo da vedere l'intera sezione delle dichiarazioni.

Esercizio (3)

Completare la Sintassi Astratta definita nel file SmallC.

SmallC: Al termine di ogni sessione

Riportare nel file SmallC.ml ogni progresso nel codice sviluppato durante la sessione.