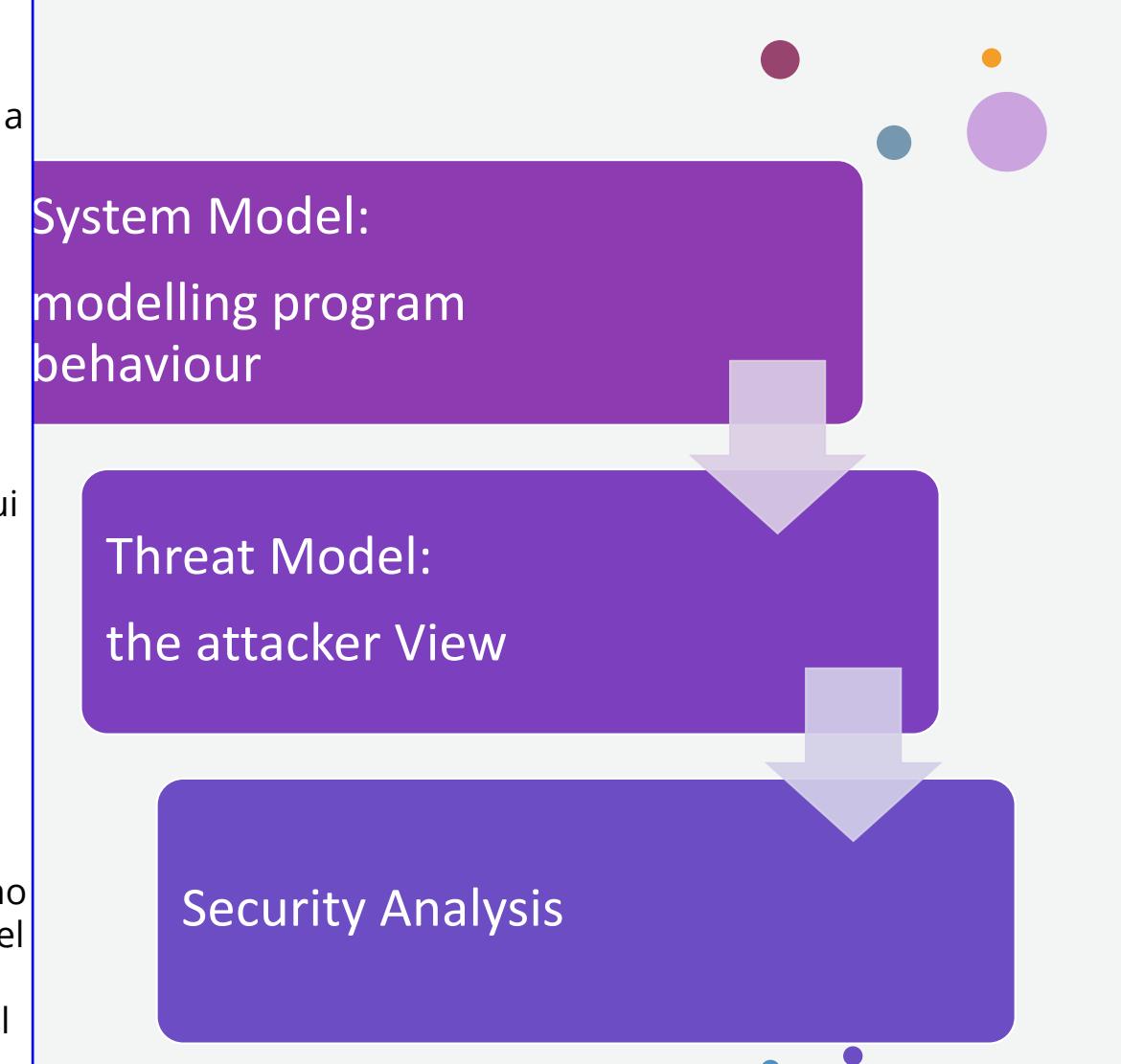




Safe Programming Languages Design and Implementation MicroFUN

Noi la volta scorsa abbiamo affrontato il problema di comprendere che cosa si intende per un linguaggio di programmazione safe e che caratteristiche deve avere. Chiaramente abbiamo affrontato soltanto uno degli aspetti della sicurezza per quanto riguarda la safeness del linguaggio di programmazione, abbiamo affrontato le problematiche relative alla memory corruption. Andremo a fare un esempio di un problema di sviluppo di un linguaggio di programmazione e analizzeremo esattamente la problematica della memory corruption e andremo ad esaminarla nel dettaglio. L'approccio che noi usiamo all'interno del corso è quello di avere tre livelli di descrizione. Il primo livello della descrizione è il modello del sistema e questo vuol dire avere dei modelli del comportamento. Abbiamo fatto l'esempio nel caso dei protocolli di sicurezza, di descrivere quali sono le azioni sui protocolli di sicurezza che vengono fatti dai principali che fanno parte del protocollo e quindi abbiamo un modello del comportamento che descrive esattamente quello che uno si aspetta dalla descrizione che viene fatta che nel caso del linguaggio di programmazione, avremo un modello del comportamento dei programmi che descrive quali sono i programmi e come vengono eseguiti. Poi abbiamo il modello dell'attaccante che ci dice il punto di vista dell'attaccante, cosa l'attaccante può fare? Quali sono le caratteristiche dell'attaccante? Abbiamo visto questo, nel caso di protocolli di crittografia, ovvero l'esempio del modello di Dolev-Yao. Che cosa può fare un attaccante nel caso di protocolli di rete e poi, una volta che abbiamo il modello di comportamento e dell'attaccante, possiamo fare l'analisi sicurezza. Allora quello che adesso noi facciamo, è farlo su un linguaggio di programmazione. Quindi definiremo il modello del comportamento del linguaggio di programmazione, vedremo il modello dell'attaccante, poi andremo a fare l'analisi della sicurezza.



System Model:
modelling program
behaviour

Threat Model:
the attacker View

Security Analysis

Modelling program behaviour

- To design a programming language (and to write programs), you have to know how the language works.
- **Semantic** analysis: The study of “how a programming language works”
- **Methods** for defining program semantics:
 - **Operational**: show how to rewrite program expressions step-by-step until you end up with a value
 - **Denotational**: how interpret a program in a different language that is well understood
 - **Axiomatic**: provide some forms of reasoning rules about programs

Allora noi useremo la semantica operazionale per discutere esempi perché ha un modo per dare un'intuizione di quello che sta succedendo e la useremo per implementare degli interpreti esattamente dei veri e propri interpreti dei linguaggi che andremo a vedere e anche per modellare il modello dell'attaccante, proprio per il fatto che è il modello dell'attaccante un modello di natura operazionale e utilizzeremo queste descrizioni per fare l'analisi.

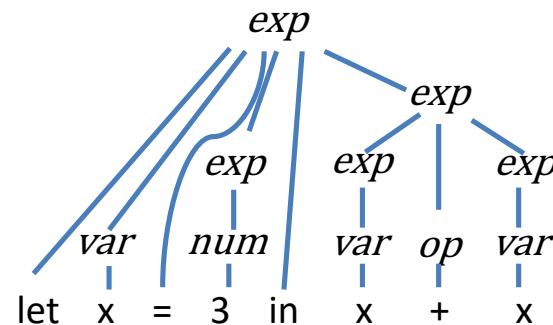
∴ Here: Operational semantics

- We'll use the operation techniques to discuss:
- *Examples* (*good for intuition*)
- Define the *implementation* (*interpreter* and *threat*) written in OCaml
- Define the *analysis*

La prima cosa che dobbiamo fare, cioè come affrontiamo il problema della sintassi, perché quando uno fa un esempio di un linguaggio di programmazione deve avere a che fare prima di tutto con la sintassi, ad esempio la sintassi del "let" di Ocaml, in cui deve essere fatto il parsing, il parsing viene fatto nella fase di analisi sintattica, viene costruito l'albero di sintassi astratta che è questo sulla destra. Non ci interessa andare a vedere come viene fatta il riconoscimento sintattico ma ci interessa comprendere come avviene il meccanismo di esecuzione. Vuol dire che noi opereremo sempre con alberi di sintassi astratta, cioè assumiamo che la sintassi e tutti i problemi relativi al riconoscimento sintattico di tutto il frontend sia già stato fatto, non ce ne preoccupiamo più.

Defining Programming Language Syntax

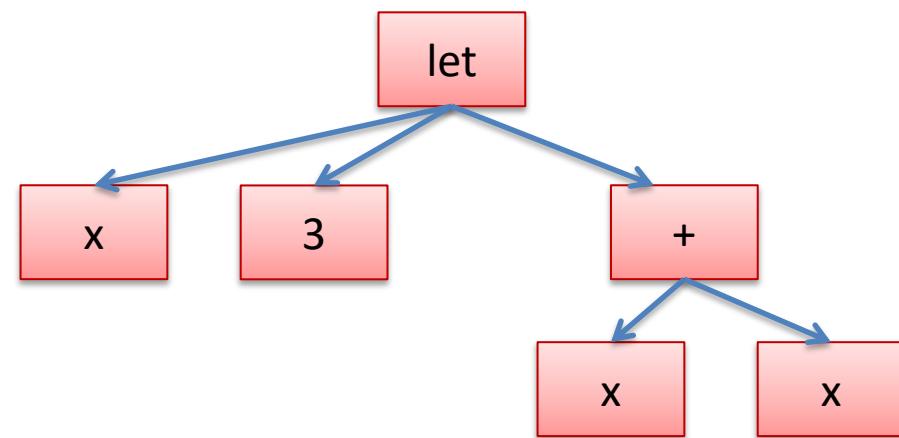
```
let x = 3 in x + x
```



This is the *parse tree*.
Useful for some purposes, but
for the semantics it's *too much information*.

Abstract Syntax Trees

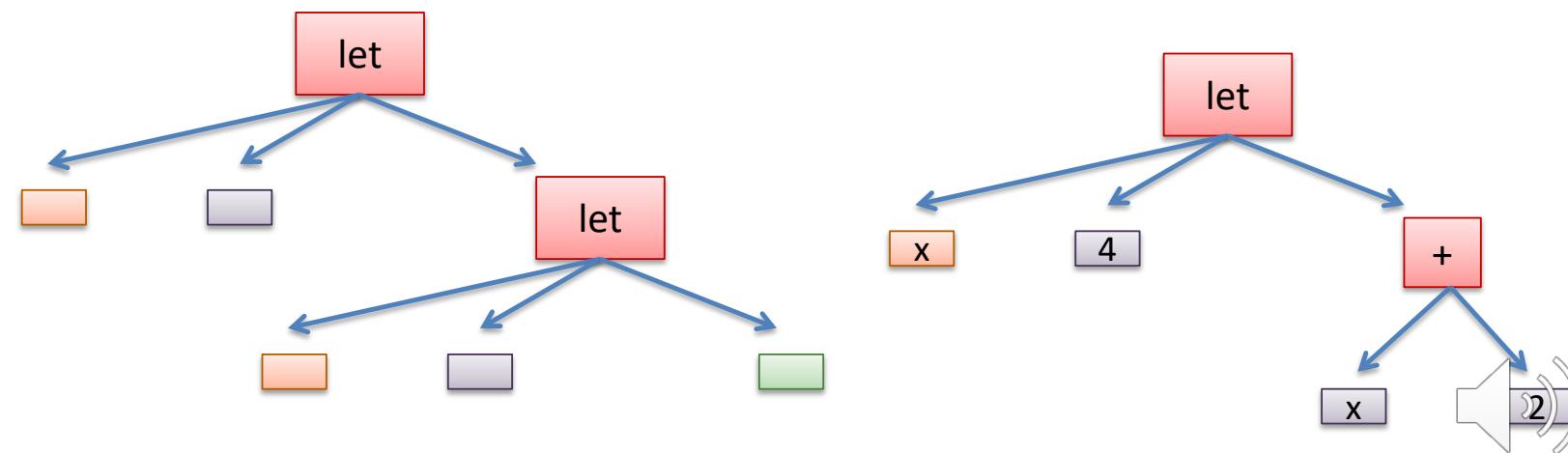
```
let x = 3 in  
x + x
```



More generally each let expression has 3 parts:

let  =  in 

And you can create complicated programs by nesting let expressions (or any other expression) recursively inside one another:



Come li definiamo gli alberi di sintassi astratta? Gli alberi di sintassi astratta vengono usati con un linguaggio funzionale il nostro caso Ocaml, che permette di fare theorem proving. Quindi all'interno di questa struttura useremo i linguaggi funzionali perché nei linguaggi funzionali viene comodo simulare quello che fa il compilatore. Quando dico simulare significa che ovviamente non ci interessa gli aspetti di efficienza e di compilazione su una particolare macchina, ma vogliamo esaminare le caratteristiche del processo di compilazione, in modo particolare della generazione del codice, e vedere come queste caratteristiche vengono fatte. Per questo motivo si trova scritto che Ocaml è un linguaggio per la programmazione funzionale, ma è sostanzialmente un linguaggio di dominio specifico per scrivere i compilatori.

OCAML

- Functional programming languages have sometimes been called “domain-specific languages for compiler writers”
- Datatypes are amazing for representing complicated tree-like structures and that is exactly what a program is.
- Idea: Use a different constructor for every different sort of expression
 - one constructor for variables
 - one constructor for let expressions
 - one constructor for numbers
 - one constructor for binary operators, like add

Allora noi dobbiamo quindi introdurre un modo strutturato per descrivere alberi di sintassi astratta e la caratteristica che ci fornisce Ocaml è abbastanza facile. Abbiamo i tipi di dati algebrici e quindi abbiamo la possibilità di codificare all'interno del linguaggio con una particolare struttura dei tipi programmata da noi che descriviamo la simulazione del processo di compilazione il meccanismo dell'albero di sintassi astratta. E qual è l'idea? Usiamo un costrutto del tipo di dato come abbiamo visto, utilizziamo un costruttore per ogni caratteristica specifica sintattica del linguaggio. Quindi, ad esempio, avremo un costruttore per le variabili, un costruttore per l'espressione di let, un costruttore per le costanti numeriche, un costruttore per le operazioni primitive binarie come add e etc... Quindi un costruttore associato alla descrizione del tipo di dato algebrico che ci permette di descrivere gli alberi di sintassi astratta del linguaggio e quindi lasciare tutti gli aspetti del frontend dei compilatori da parte.

Andiamo a vedere come costruiamo gli alberi di sintassi astratta ed è per questo che è stato chiamato MicroFun, un micro linguaggio di programmazione funzionale. Allora definiamo la categoria sintattica delle espressioni che sono esattamente le espressioni che costituiscono i programmi scritti in questo linguaggio di programmazione. Andiamo a vedere usando questa strategia, cioè di costruire un tipo di dato algebrico che rappresenta, tramite opportuni costruttori, la sintassi del linguaggio in termini degli alberi di sintassi astratta del linguaggio e lo definiamo questo modo. Abbiamo il tipo delle espressioni, abbiamo i valori numerici primitivi. In questo caso i valori numerici primitivi sono le costanti intere e quindi definiamo al costruttore CstI che si trasporta un valore intero, le costanti booleane e quindi definiamo la costante CstB che vuol dire che trasporta una costante booleana. Poi abbiamo le variabili, identificatori presenti nel programma.

(OCAML) AST for MicroFUN

```
type expr =  
| CstI of int  
| CstB of bool  
| Var of string  
| Let of string * expr * expr  
| Prim of string * expr * expr  
| If of expr * expr * expr  
| Fun of string * expr (* Lambda , param Body *)  
| Call of expr * expr
```

Abbiamo il let, che prende una stringa che sono la variabile che stiamo definendo, l'espressione che una volta valutata sarà il valore che viene associata alla variabile introdotta nel let e poi il corpo dell'espressione, quindi il let esattamente quella cosa che abbiamo visto prima, è un costruttore di tipo che costruisce un alberino che ha 3 sottoalberi: l'identificatore, il valore della espressione che, una volta valutata, darà il valore alla variabile che viene prodotta e il corpo del let. Poi abbiamo l'IfThenElse che ha un'espressione che è la guardia, un'espressione che è il ramo then e l'altra espressione che è il ramo else. Infine abbiamo le lambda, abbiamo delle funzioni anonime, quindi il fatto che io posso definire una funzione che prende un parametro, un corpo. Come faccio a dare un nome a questa espressione? Uso il let quindi non c'è bisogno di introdurre una nozione di funzione in cui definisco anche il nome della funzione, questo lo posso fare esattamente usando il let. Notate che abbiamo adottato la stessa strategia di Ocaml.

Se non scriviamo le lambda in questo modo, queste funzioni non sono ricorsive, quindi in questo momento stiamo considerando un linguaggio di programmazione funzionale che sono esattamente i lambda presenti nei linguaggi a oggetti che non sono ricorsive, che sono delle funzioni anonime a cui noi possiamo associare un nome tramite il let. Infine abbiamo la call, l'applicazione funzionale, cioè la call prende due espressioni, la prima dovrà restituire una funzione, sia essa una funzione a cui viene associato un nome, un let, sia essa una espressione che è una funzione anonima, quindi un tipico lambda e una volta che abbiamo valutato questa espressione, quindi usiamo la caratteristica dell'operazione del meccanismo funzionale dove le funzioni sono entità di ordine superiore, quindi posso tenere e sono valori, quindi le posso tenere anche come risultato della valutazione di un'espressione, posso passare l'argomento, in questo caso stiamo considerando unario e vedete qui nella fun prendiamo soltanto un parametro, ma possiamo benissimo estenderlo a 2-3 parametri. In questo caso l'espressione del sotto albero della call corrisponde all'espressione del parametro attuale che dovrà essere valutata per ottenere un valore che poi dovrà essere associato al nome del parametro formale. Allora questi sono i nostri alberi di sintassi astratta.

```
type expr =  
| CstI of int  
| CstB of bool  
| Var of string  
| Let of string * expr * expr  
| Prim of string * expr * expr  
| If of expr * expr * expr  
| Fun of string * expr (* Lambda , param Body *)  
| Call of expr * expr
```

(OCAML) AST for

```
let elambda = Fun("x", Prim("+", Var "x", CstI 1));;
```



```
fun x = x+1;;
```

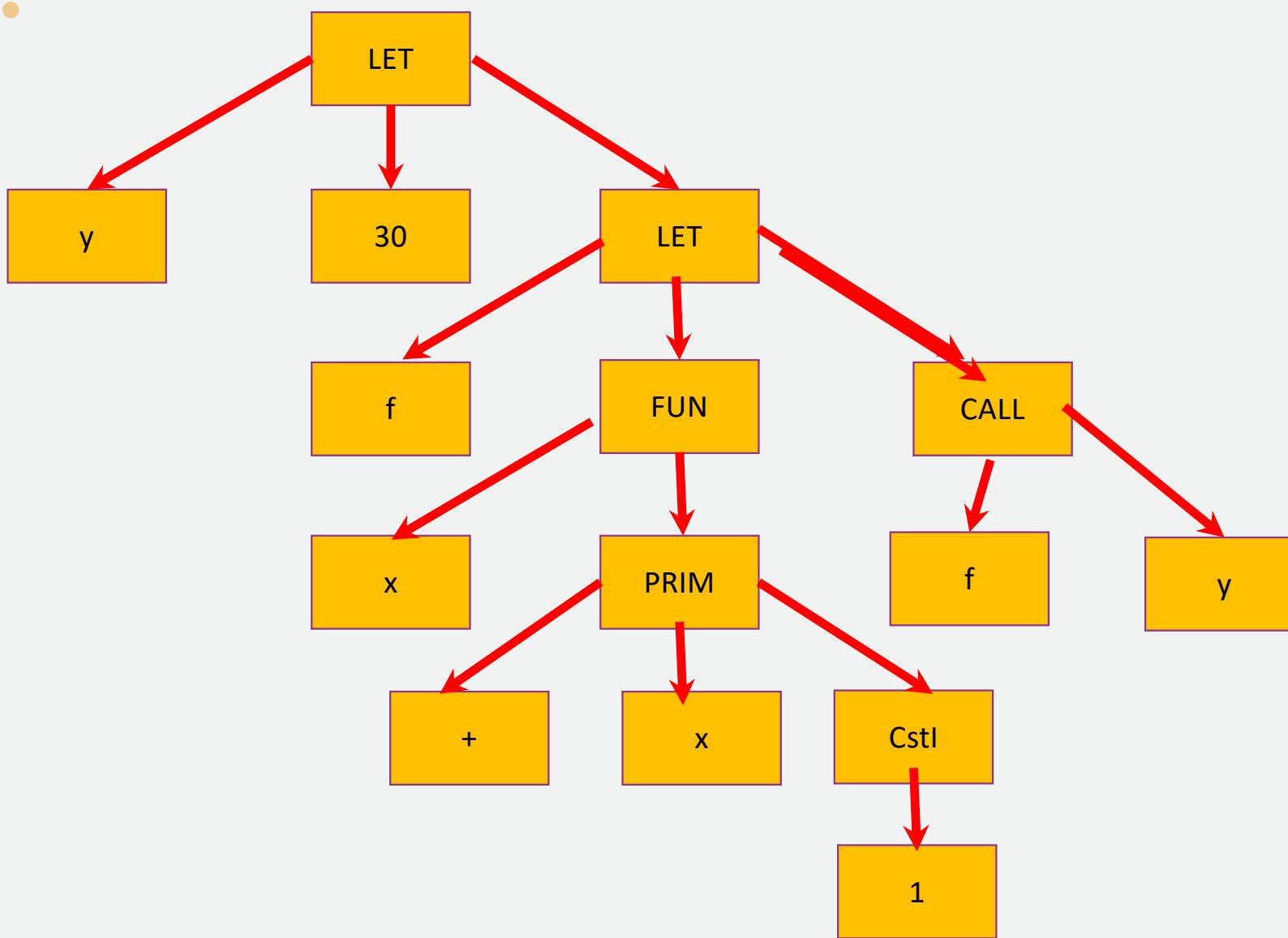
Allora uno può scrivere questa cosa, può usare la caratteristica repl di Ocaml e quindi, ad esempio dire io ho un'espressione lambda che è una fun che prende un parametro X, fa poi un'operazione primitiva che è una somma, ha al suo interno l'occorrenza della variabile x e poi questa variabile x viene incrementata di uno perché stiamo facendo un'operazione di costante intera. Allora questa è l'espressione che corrisponderebbe a fare $x = x + 1$, ma è molto più verbosa perché è un'espressione in sintassi astratta. Se uno lo scrive sempre per far vedere qual è la differenza tra la sintassi astratta e la sintassi concreta in termini di alberi di sintassi astratta che stiamo considerando,

```
let y = 30 in  
  let f = fun x = x+1 in  
    f y;;
```

CONCRETE SYNTAX

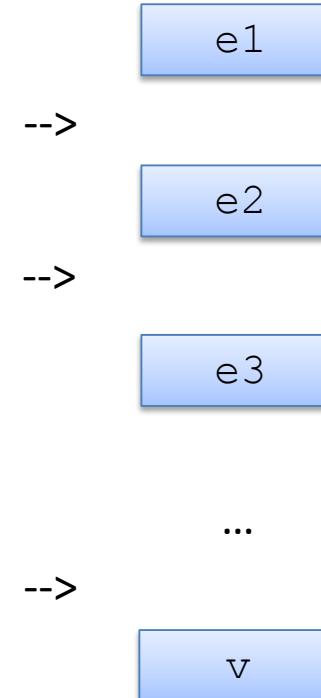
```
Let ("y" , CstI 30,  
     Let("f", Fun("x", Prim("+", Var "x", CstI 1)),  
          Call(Var "f", Var "y")));;
```

ABSTRACT SYNTAX – ABSTRACT SYNTAX TREE



Allora com'è fatta la valutazione? La valutazione abbiam detto sarà un qualcosa che prende l'albero di sintassi astratta, opera sull'albero di sintassi astratta, fa delle trasformazioni sull'albero di sintassi astratta fin tanto che non produce un valore, un valore che siamo in un linguaggio funzionale. Ovviamente se fossimo in un linguaggio imperativo dovrebbe produrre una variazione di stato, se fosse un linguaggio oggetti dovrebbe dire come cambiano gli oggetti. Nel nostro caso, sostanzialmente valutare significa fare una visita dell'albero di sintassi astratta, tenendo conto delle regole di precedenza degli operatori, fintanto che non si produce un valore, quindi un'azione che riscrive sostanzialmente l'albero di sintassi astratta fino a ottenere un valore.

EVALUATION



Evaluation rewrites the abstract syntax of a program step-by-step until it produces a value.

```
let x = 30 in  
let y = 20 + x in  
x+y
```

-->

```
let y = 20 + 30 in  
30+y
```

-->

```
let y = 50 in  
30+y
```

-->

```
30+50
```

-->

```
80
```

evaluation complete: we have produced a *value*

RUN TIME STRUCTURE (STACK)

An environment is a map from identifier to a value (what the identifier is bound to). We represent the environment as an association list, i.e., a list of pair (identifier, data).

```
type 'v env = (string * 'v) list
```

Given an environment {env} and an identifier {x} it returns the data {x} is bound to. If there is no binding, it raises an exception.

```
let rec lookup env x =
  match env with
  | []    -> failwith (x ^ " not found")
  | (y, v)::r -> if x=y then v else lookup r x
```

A questo punto, dobbiamo dire quali sono i valori a run time, i valori che rappresentano il risultato della valutazione dei programmi. Noi stiamo considerando un linguaggio di programmazione con scoping statico. Lo scoping statico, a differenza dello scoping dinamico, definisce le regole di visibilità dei nomi in base alla struttura sintattica del programma, ovvero quando io dichiaro una funzione, questa funzione avrà associato un ambiente di visibilità al momento in cui questa funzione è stata dichiarata. Allora a quel punto, le regole di visibilità dei nomi che compaiono nel corpo della funzione dipendono esattamente non dall'ambiente in cui questa funzione verrà valutata tramite una call, ma dipendono soltanto dalla struttura sintattica del programma.

∴ RUN TIME VALUES

A runtime value is an integer or a function closure
Boolean are encoded as integers.

```
type value =  
| Int of int  
| Closure of string * expr * value env
```



THE INTERPRETER

Operational rules

$$\frac{env(x) = v}{env \triangleright Var\ x \Rightarrow v}$$

eval (Var x) env -> lookup env x

$$\frac{env \triangleright e1 \Rightarrow v1 \quad env \triangleright e2 \Rightarrow v2}{env \triangleright \text{Prim}(+,e1,e2) \Rightarrow v1 + v2}$$

eval Prim("+", e1, e2) env ->
eval e1 env + eval e2 env

L'approccio che noi seguiamo, che è un approccio standard, è quello di dire che ci sono delle regole operazionale che corrispondono alle regole dell'interprete. Se io voglio andare a vedere qual è il valore associato alla variabile x, l'operazione che devo fare è andare a vedere nell'ambiente e vedere se nell'ambiente è associato a v. Quindi questa è la regola mi dice: sto in uno stato che ho l'ambiente.

L'ambiente lo potete immaginare come il run time stack e trovo una variabile quando sto eseguendo. Allora devo andare a vedere com'è fatto il legame di ambiente. Quindi se io chiamo eval, il nome dell'interprete, quando eval viene applicato con un certo valore di run time dell'ambiente e la struttura sintattica che è l'occorrenza della variabile x, l'unica cosa che deve fare è chiamare l'istruzione del run time support, che è l'operazione di lookup che va a vedere qual è il valore dell'ambiente della variabile x. Allora in termini di strutture di implementazione concreta, cosa corrisponde? L'ambiente e lo stack. La x è l'index della variabile, a questo punto vado a vedere il valore dell'ambiente corrente, l'indirizzo di base del record di attivazione corrente, gli applico l'offset e ottengo il valore della x. Similmente, con le operazioni primitive, immaginate di avere l'operazione primitiva e state valutando nel vostro interprete l'operazione primitiva e gli diciamo che quella è un'operazione primitiva plus, che prende due espressioni nell'ambiente corrente. Allora cosa fa? Questo qui è uno stile di programmazione eager, valutate l'ambiente, l'espressione uno, valutate l'espressione due, quindi prima valutiamo gli argomenti e poi abbiamo la funzione primitiva "più" in questo caso, è la funzione primitiva più di Ocaml che ci fa la somma di interi e quindi il risultato sarà la somma della valutazione dei due, quindi vedete quello che facciamo, andiamo sull'albero di sintassi astratta e facciamo una discesa sull'albero, operiamo per costruire i comportamenti legali.

Let

$$\frac{env \triangleright erhs \Rightarrow xval \quad env[xval/x] \triangleright ebody \Rightarrow v}{env \triangleright \text{Let } x = erhs \text{ in } ebody \Rightarrow v}$$

```
eval (Let(x, erhs, ebody)) env ->
    let xval = eval erhs env in
        let env1 = (x, xval) :: env in
            eval ebody env1
```

Il let come ha fatto? Ha la dichiarazione di una variabile, l'espressione che corrisponde al valore che poi dovremmo dare alla variabile e il corpo del let in cui che dovrà essere valutato tenendo conto del legame di ambiente tra x e il valore dell'espressione erhs. Questo quand'è che ci darà v? Ci darà v se xval è il valore dell'espressione che viene introdotta con il let nell'ambiente corrente. A questo punto dobbiamo estendere l'ambiente corrente, con il legame tra x che è la variabile che introduciamo con il let e il valore xval, che è il valore che abbiamo ottenuto andando a valutare l'espressione dichiarata nel let. Una volta fatta questa estensione di ambiente, quindi di fatto cosa stiamo facendo? Un'operazione di push sullo stack del nuovo record di attivazione che contiene il legame tra la variabile che abbiamo dichiarato con il let e il valore dell'espressione che abbiamo dichiarato con il let. Una volta fatto questo, possiamo valutare il corpo del let, se il corpo del let da un valore, a questo punto il valore è il risultato.

Let & run time stack

env =
run-time
stack

push RA env

$$\frac{env \triangleright e1 \Rightarrow v1 \quad env[v1/x] \triangleright e2 \Rightarrow v2}{env \triangleright Let(x, e1, e2) \Rightarrow v2}$$

pop RA env

```
let rec eval (e : expr) (env : value env) : value =
  match e with
  | CstI i -> Int i
  | CstB b -> Int (if b then 1 else 0)
  | Var x -> lookup env x
  | Prim(ope, e1, e2) ->
    let v1 = eval e1 env in
    let v2 = eval e2 env in
    begin
      match (ope, v1, v2) with
      | ("*", Int i1, Int i2) -> Int (i1 * i2)
      | ("+", Int i1, Int i2) -> Int (i1 + i2)
      | ("- ", Int i1, Int i2) -> Int (i1 - i2)
      | ("=", Int i1, Int i2) -> Int (if i1 = i2 then 1 else 0)
      | ("<", Int i1, Int i2) -> Int (if i1 < i2 then 1 else 0)
      | _ -> failwith "unknown primitive or wrong type"
    end
```



- | Let(x, eRhs, letBody) ->
let xVal = eval eRhs env in
let letEnv = (x, xVal) :: env in
eval letBody letEnv
- | If(e1, e2, e3) ->
begin
match eval e1 env with
| Int 0 -> eval e3 env
| Int _ -> eval e2 env
| _ -> failwith "eval If"
end

Che cosa succede quando nell'ambiente corrente trovo una lambda? Quando trovo una lambda devo costruire la chiusura. Sono nell'ambiente env, quindi vuol dire questo è l'ambiente al momento della dichiarazione, com'è fatta la chiusura? La chiusura dice, io metto il nome del parametro formale x e il body e a quel punto gli metto il puntatore all'ambiente corrente che è esattamente l'env dove è stato dichiarato. Se vado tra un'occorrenza della "f" e mi dice che questa occorrenza della "f" è associata a una chiusura e la chiusura è fatta in questo modo, allora quando vado alla call vado a vedere il valore associato alla "f", vado a vedere il valore associato alla chiusura e a quel punto cosa vado a fare? Vado ad estendere l'ambiente al momento della dichiarazione. Dove trovo l'ambiente al momento della dichiarazione? È nella chiusura associata alla funzione, quindi dove vado a valutare il corpo della funzione? Vado a valutare il corpo della funzione nell'ambiente al momento della dichiarazione esteso con il legame tra il nome del parametro formale x e il valore del parametro attuale. Vedete che prima vado a valutare il valore del parametro attuale "arg" nell'ambiente corrente e se il corpo della funzione mi da v, il risultato della funzione è v. Cosa vuol dire questo? Quando faccio una call di una funzione, vado a valutare l'argomento nell'ambiente del chiamante, creo il record di attivazione della funzione, dell'attivazione di questa funzione che contiene il legame tra cosa? Tra il nome del parametro formale x e il valore del parametro attuale.

Ma il body della funzione non lo vado a valutare nell'ambiente corrente esteso con il nuovo legame di ambiente, ma la vado a valutare nell'ambiente che avevo al momento della dichiarazione, perché sono con scoping statico, ovvero tutti i riferimenti non locali devo risolverli come se fossi a dichiarazione, quindi tecnicamente devo usare l'ambiente di dichiarazione della funzione. In termini di strutture di implementazione, di fatto quello che stiamo utilizzando è lo static link, che un'informazione che è presente nel record di attivazione. Quindi uso lo static link, vado a vedere come è fatto il puntatore all'ambiente a quel punto uso quel valore per poter risolvere i riferimenti non locali. Se avessi avuto un linguaggio di programmazione con scoping dinamico, semplicemente qui non avrei avuto la chiusura, ma avrei avuto l'ambiente corrente perché a quel punto sarei andato ad esaminare i legami di ambiente soltanto nel flusso di esecuzione corretto.

$$env \triangleright Fun(x, e) \Rightarrow Closure("x", e, env)$$

$$\frac{env \triangleright Var("f") \Rightarrow Closure("x", body, fDecEnv) \quad env \triangleright arg \Rightarrow va \quad fDecEnv[v^a/x] \triangleright body \Rightarrow v}{env \triangleright Call(Den("f"), arg) \Rightarrow v}$$

```
| Fun(x,fBody) -> Closure(x, fBody, env)
| Call(eFun, eArg) ->
  let fClosure = eval eFun env in
  begin
    match fClosure with
    | Closure (x, fBody, fDeclEnv) ->
        let xVal = eval eArg env in
        let fBodyEnv = (x, xVal) :: fDeclEnv
        in eval fBody fBodyEnv
    | _ -> failwith "eval Call: not a function"
  end
```

The attacker

- The attacker can corrupt the memory!!
- Question: is our language implementation memory safe?

A questo punto dobbiamo definire l'attaccante. Quindi abbiamo fatto il primo passo. Il modello del threat è può corrompere la memoria, ovvero può fare delle operazioni che corrompono la memoria. La nostra implementazione che abbiamo visto di questo linguaggio semplice è un implementazione che garantisce che la memoria non viene corrotta? Detto in altri termini, il linguaggio che abbiamo considerato in questo momento è un safe programming language?

Dovrebbe esserlo perché l'utente non può gestire la memoria direttamente, quindi visto che noi simuliamo il runtime con questo meccanismo dell'ambiente, l'utente non ha modo di fare operazioni complicate sugli stack, quindi fare delle operazioni sullo stack, quindi a questo punto non può corrompere la struttura dello stack. Eh, è vero, in questa simulazione questo tipo di problematiche non vengono create, quindi da questo punto di vista è safe. Adesso però vi chiedo, è l'unico problema che noi abbiamo in questo linguaggio di memory corruption, cioè operare sullo stack o abbiamo altri problemi di memory corruption? Allora la type confusion tra Int e Bool la risolviamo a livello del typechecking. Quando assumiamo l'int di un intero prima di andarlo poi a valutarlo successivamente, non controlliamo la dimensione massima che questo possa avere. Ci fidiamo soltanto sul fatto che l'operazione primitiva di ocaml fa la cosa per noi. Quando noi definiamo la costante intera, gli mettiamo un controllo, però lì è sintassi, quindi, essendo sintassi chi lo dovrebbe controllare? Il run time o il frontend. Essendo sintassi front end, quindi qui siamo a run time, quindi vuol dire che se siamo arrivati lì, vuol dire che è già un intero rappresentabile. quindi non è lì il punto. Oppure ad ogni entry nel match, anziché fare direttamente $i1 * i2$, ad esempio, bisogna fare un controllo che $i1$ sia inferiore della radice quadrata del valore massimo, stessa cosa per $i2$ e una volta che si è verificato che la moltiplicazione fra $i1$ e $i2$ non può dar luogo a un overflow, a quel punto possiamo eseguirlo. Allora innanzitutto bisogna metterci delle informazioni che sono informazioni di run time in modo particolare, bisogna mettere in tutta l'implementazione delle operazioni primitive, il fatto che noi abbiamo un max int, il massimo intero rappresentabile. Bisogna controllare che $i1$ e $i2$ siano minori di max int. Perché se io rappresento sintatticamente, rappresento un valore intero, però a questo punto, una volta che faccio il controllo, che non siano interi rappresentabile, dove evitare che la somma o il prodotto non vadano fuori della dimensione. Vuol dire che tutti i meccanismi che hanno a che vedere con l'implementazione delle funzioni primitive, in questa nostra implementazione, sono memory unsafe tutti. Se avessi voluto introdurre i numeri naturali? E un po' di operazioni primitive sui numeri naturali. Cosa avrei dovuto fare? Forse un primo passo sarebbe quello di decidere quale debba essere la semantica delle operazioni fra naturali, cioè se devono essere tutte operazioni interne oppure se si ammettono operazioni che possono andare al di fuori, andare fuori del tipo naturali. E quindi nel caso specifico della sottrazione, decidere se è accettabile sottrarre un naturale maggiore da un naturale minore, e quindi "scappare" negli interi. Però se uno usa uno stile funzionale, secondo lei questo succede? E chi lo decide a questo? Lo si definisce in base all'operazione primitiva, tipicamente nei linguaggi di programmazione funzionale come quello, ad esempio, vedere qui usiamo lo star, ma se volessimo usare gli star sui float avremmo dovuto mettere il punto, quindi in questo caso lo stare è un'operazione che prende $\text{int} \rightarrow \text{int}$. Il float avrebbe preso $\text{float} \rightarrow \text{float} \rightarrow \text{float}$ quindi avremo tutte le operazioni definite per il tipo degli elementi. A quel punto, una volta definita l'analisi dei tipi relativamente alle operazioni primitive, poi a quel punto vanno fatti i controlli per evitare che ci siano degli escape dei tipi.