The Grammar
Declarations-Commands

The Imperative Language Simple

[0]**P**rogram= **D**eclaration **C**ommands **|** [1]**C**ommands

[2]**D**::= **ide O**Theridentifiers **;**

[3]**O**::= **ide O |** [4] ε

[5]**Cs**::= **C**ommand **; Cs |** [6] ε

[7]**C**::= **A**ssign | [8] **W**hile

[9]**A**::= **ide :=** **E**xpression

[10]**W**::= **while E do C Cs endwhile**

The Grammar
Expressions

[11]**E::= F E'**

[12]**E'::= op-l**ower **F E' |** [13] ε

[14]**F::= T**erm **F'**

[15]**F'::= op-h**ight **T F' |** [16] ε

[17]**T::= num |** [18] **ide |** [19] **(E)**

Correlated Occurrences

1) All the used identifiers are correctly declared

2) All ther declared identifier are used

3) All the variables have an assigned values before the use

4) Iterator guard expressions have type boolean

All the used identifiers are correctly declared
Choice of the attributes

Attribute Plan: names and properties of the attributes

u: - set of the used identifiers
- **synthesized of Su={Cs,C,A.W,E,E',F,F',T}**
- **$\forall X \in Su$, X.u=I iff $X \Rightarrow^* \alpha \equiv \alpha_1 \ldots \alpha_n \in \Sigma^*$ and if α_i=ide then α_i.lexeme$\in$I**

d: - set of the declared identifiers -
- **synthesized of Sd={D,O}**
- …

r: - set inclusion of the used into the declared ones
- s**ynthesized of {P}**
- **r=u≤d**

Values and auxiliary functions that are used in the actions

Set are handled by list with the following operations

cons: ide X ide-list --> ide-list
emptylist : --> ide-list
append: ide-list X ide-list --> ide-list
included: ide-list X ide-list --> boolean
isempty: ide-list --> boolean

E'::= ε

[0] **P::= D Cs**	**P.r:= include(Cs.u, D.d)**
[1] **P::= Cs**	**P.r:= isempty(Cs.u)**
[2] **D::= var ide O**	**D.d:= cons(ide.lexeme, O.d)**
[3] **O_1::= , ide O_2**	**O_1.d::= cons(ide.lexeme, O_2.d)**
[4] **O::=ε**	**O.d::= emptylist**
[5] **Cs_1::= ; C Cs_2**	**Cs_1.u:= app(C.u, Cs_2.u)**
[6] **Cs::= ε**	**Cs.u:= emptylist**
[7] **C::= A**	**C.u:= A.u**
[8] **C::=W**	**C.u:=W.u**
[9] **A::= ide := E**	**A.u:= cons(ide.lexeme, E.u)**
[10] **W::=while E do C Cs edw**	**W.u:= app(E.u, app(C.u, Cs.u))**
[11] **E::= F E'**	**E.u:= app(F.u, E'.u)**
[12] **E'_1::= op-l F E'_2**	**E'_1.u:= app(F.u, E'_2.u)**
[13] **E::= ε**	**E.u:= emptylist**
[14] **F::= T F'**	**F.u:= app(T.u, F'.u)**
[15] **F'_1::= op-hight T F'_2**	**F'_1.u:= app(T.u, F'_2.u)**
[16] **F'::= ε**	**F'.u:= emptylist**
[17] **T::= num**	**T.u:= emptylist**
[18] **T::= ide**	**T.u:= cons(ide.lexeme, emptylist)**
[19] **T::= (E)**	**T.u:= E.u**

All the variables are correctly initialized before the use

uin: - set of the variables that have been assigned to, in the sequence that precedes the current statement
- **inherited of Su={Cs,C,A.W,E,E',F,F',T}**
- **....**

uout: - set of the variables assigned to, in the sequence ended by the current statement
- **synthesized of Sd={Cs,C,A.W,E,E',F,F',T}**
- **....**

r: - predicate rhat holds if the used ave been previously assigned to
- s**ynthesized of all the program structures, but declarations, {P,C,Cs,A,W,E,E',...}**
- **...**

Values and auxiliary functions that are used in the actions

Set are handled by list with the following operations

cons: ide X ide-list --> ide-list

emptylist : --> ide-list

append: ide-list X ide-list --> ide-list

included: ide-list X ide-list --> boolean

isempty: ide-list --> boolean

[0]**P::= D Cs**	**P**.r:= **Cs**.r , **C**s.uin:=emptylist
[1]**P::= Cs**	**P**.r:= **Cs**.r , **C**s.uin:=emptylist
[2]**D::= var ide O** [3]**O1::= , ide O2** [4]**O::=ε**	?
[5]**Cs1::= ; C Cs2**	**Cs1**.r:=(**C**.r&**Cs2**.r), **C**.uin:=**Cs1**.uin **C**s1.uout:=**C**s2.uout,**C**s2.uin:=**C**.uout
[6]**Cs**::= ε	**Cs**.r:= true,**C**s.uout:=**C**s.uin
[7]**C::= A**	**C**.r:= **A**.r, **A**.uin:=**C**.uin, **C**.uout:=**A**.uout
[8]**C::=W**	**C**.r:= **W**.r, **W**.uin:=**C**.uin, **C**.uout:=**W.out**
[9]**A::= ide := E**	**A**.r:= **E**.r, **E**.uin:=**A**.uin, **A**.uout:=cons(**ide**.lexeme,**A**.uin)
[10]**W::= while E do C endw**	**W**.r:= (**E**.r & **C**.r), **E**.uin:=**W**.uin, **C**.uin:=**W**.uin,**W**.uout:=**C**.uout
[11]**E::= F E'**	**E**.r:= (**F**.r & **E'**.r), **F**.uin:=**E**.uin, **E'**.uin:=**E**.uin,
[12]**E'1::= op-l F E'2** [13]**E**::= ε [14]**F::= T F'** [15]**F'1::= op-h T F'2**	?
[16]**F'**::= ε	**F'**.r:= true
[17]**T::= num**	**T**.r:= true
[18]**T::= ide**	**T**.r:= isin(**ide**.lexeme,**T**.uin)
[19]**T::= (E)**	**T**.r:= **E**.r, **E**.uin:= **T**.uin

C.uout = emptylist

W.uout = W.uin

To Be Completed

Type Checking (1)
A Case Analysis

- Extending the language with basic types: A Grammar
- Planning Type Analysis: Updating Symbol-Table
- Inheriting the list of the variables of a given type
- Using a different grammar:
  - Inheriting the type of a given list: But (… is it an L-attributed grammar?)
  - Only sinthesized attributes: Is it possible?

An LL(1) Grammar for Simple extended with basic types

[0]**Program= Declarations Commands** | [1]**Commands**

[2]**Ds::= Var Dtypeds**

[3]**Dts::= Dt Dts'**

[4]**Dts' ::= ; Dt Dts'** | [5] ε

[6]**Dt::= ide Otheridentifiers**

[7]**O::= , ide O** | [8] **:** t**Y**pe

[9]**Cs::= ; Command Cs** | [10] ε
[11]**C::= Assign** | [12] **While**
[13]**A::= ide := Expression**
[14]**W::= while E do C Cs endwhile**

[15]**E::= F E'**
[16]**E'::= op-lower F E'** | [17] ε
[18]**F::= Term F'**
[19]**F'::= op-hight T F'** | [20] ε
[21]**T::= num** | [22] **ide** | [23] **(E)**

[24]**Y::= boolean** | [25] **integer** | [26] **file** | ...

Updating Symbol-Tables with Types for Variables

We are dealing with side-effects (SDD):

- Modifications of the Symbol-Table: adding types
- Operation on symbol table: *Addtype*: entry X type

Attributes:

entry:- row of the symbol table
-synthesized of the program variables, **ide**

t : -type expression
-synthesized of type annotation, **Y**

ty : -list of the entries
-inherited of variable declaration, **Dt,Dts, Dts', O**

To Be Completed

Noting the use of the iterator-based action

Updating Symbol-Tables with Types for Variables Another Grammar…

…that avoids the use of the iterator-based action.

Attributes:

entry:- row of the symbol table
-synthesized of the program variables, **ide**

t : -type expression
-synthesized of type annotation, **Y**

ty : -Type expression
-inherited of variable declarations, **Dt,Dts, Dts', O**

[0] **P::= Ds Cs**	
[1] **P::= Cs**	?
[2] **Ds::= Var Dts**	
[3] **Dts::=Dt : Y Dts'**	**Dt.ty:= Y.t**
[4] **Dts' ::= ; Dt : Y Dts'**	**Dt.ty:=Y.t**
[5] **Dts' ::= ε**	
[6] **Dt::=ide O**	**O.ty:= Dt.ty, addtype(ide.entry,Dt.ty)**
[7] **O_1::= , ide O_2**	**O_2.ty:=O_1.ty, addtype(ide.entry,O_1.ty)**
[8] **O::= ε** [9] **Cs_1::= ; C Cs_2** [10] **Cs::= ε** [11] **C::= A** [12] **C::=W** [13] **A::= ide := E** [14] **W::=while E do C Cs edw** [15] **E::= F E'** [16] **E'::= op-l F E'** [17] **E'::= ε** [18] **F::= T F'** [19] **F'::= op-h T F'** [201] **F'::= ε** [21] **T::= num** [22] **T::= ide** [23] **T::= (E)**	
[24] **Y::= boolean**	**Y.t:= boolean**
[25] **Y::= integer**	**Y.t:= integer**
[26] **Y::= file**	**Y.t:= file**

But
is the preceding one, an L-attributed Grammar ?

No of course, since in production [4], Dt inheriths from its right brother Y.

A different use of the attributes

Attributes:

entry:- list of the symbol table rows of the variables
- synthesized of variable declarations: **Dt, O**

t : -type expression
-synthesized of type annotation, **Y**

Operation on symbol table: *Addtype-set*: list-of-entries X type

[0] **P::= Ds Cs**	?
[1] **P::= Cs**	
[2] **Ds::= Var Dts**	
[3] **Dts::=Dt : Y Dts'**	addtype-set(Dt.entry,Y.t)
[4] **Dts' ::= ; Dt : Y Dts'**	addtype-set(Dt.entry,Y.t)
[5] **Dts' ::= ε**	
[6] **Dt::=ide O**	Dt.entry:= cons(ide.entry,O.entry)
[7] **O_1::= , ide O_2**	O_1.entry:=cons(ide.entry,O_2.entry)
[8] **O::= ε**	O.entry:=emptylist
[9] **Cs_1::= ; C Cs_2**	
[10] **Cs::= ε**	
[11] **C::= A**	
[12] **C::=W**	
[13] **A::= ide := E**	
[14] **W::=while E do C Cs edw**	
[15] **E::= F E'**	
[16] **E'::= op-l F E'**	
[17] **E'::= ε**	
[18] **F::= T F'**	
[19] **F'::= op-h T F'**	
[20] **F'::= ε**	
[21] **T::= num**	
[22] **T::= ide**	
[23] **T::= (E)**	
[24] **Y::= boolean**	Y.t:= boolean
[25] **Y::= integer**	Y.t:= integer
[26] **Y::= file**	Y.t:= file

Comparing Solutions

First and second solutions are similar:

- Both are using one inherithed attribute;

But they differ for:

- The first one is L-attributed whilst second one is not;
- The first one is using one iterator-based action whilst second one is not;

Hence, third solution seems the best one, since: they differ for:

- It L-attributed and also S-attributed
- It is not using any iterator-based action.

In fact, this is not the case, since:

- addtype-set is a masking of an iterator-based action by using a set operator
- Next slide contains the definition of one 2-attributed, S-attributed, Grammar that is:
    - + a Variant of the first one
    - + it is really, not using iterator-based actions

A Variant of the first Grammar

Updating Symbol-Tables with Types for Variables

Attributes:

entry:- row of the symbol table
 -synthesized of the program variables, **ide**

t : -type expression
 -synthesized of **Y, O**

. . .	
[6] **Dt**::= **ide O**	addtype(Ide.entry,**O**.t)
[7] $\mathbf{O_1}$::= **, ide** $\mathbf{O_2}$	addtype(Ide.entry,$\mathbf{O_2}$.t); $\mathbf{O_1}$.t::= $\mathbf{O_2}$.t
[8] **O**::= **: Y**	**O**.t::= **Y**.t
. . .	
[24] **Y**::= **boolean**	**Y**.t::= boolean
[25] **Y**::= **integer**	**Y**.t::= integer
[26] **Y**::= **file**	**Y**.t::= file

Concluding Remarks

Choose Syntactic Grammar carefully

Different Syntactic Grammars lead to attribute plans that differ one another for: number and properties of the used attributes

Some Syn. Grammar allow more plans than other and with different difficulty levels

Choose the Plan carefully

Given an LL (LR) grammar, non L-attributed plans can be found before to realize how to define one plan that is L-attributed

Type Checking (2)

- **Associating Types to Program Structures: Typing Rules**
- **Extending the attribute plans of the previous Grammars**
- Derived Types: Typing Rules
- Coercion and overloading: Typing Rules
- Extending the attribute plans of the previous grammars

Assigning Types to Program Structures of a Strongly Typed Language

Types

Types=Basic-Types +N +Types X Types ->Types

Rules

$$\frac{\Gamma \vdash i:t \quad \Gamma \vdash e:T}{\Gamma \vdash i:=e:N} \qquad \frac{\Gamma \vdash e:boolean \quad \Gamma \vdash Cs: N}{\Gamma \vdash \text{while } e \text{ do } Cs :N} \qquad \frac{\Gamma \vdash C:N \quad \Gamma \vdash Cs:N}{\Gamma \vdash C;Cs : N}$$

$$\frac{}{\Gamma_1,i:T,\Gamma_2 \vdash i:T} \qquad \frac{\Gamma \vdash e1:T1 \quad \Gamma \vdash e2: T2 \quad \Gamma \vdash op:T1xT2\text{->}T}{\Gamma \vdash op(e1,e2):T} \qquad \frac{\Gamma,i:Ti,j:Tj \vdash e:T}{\Gamma \vdash fun(I,j)\{e\} : TixTj\text{->}T}$$

Attributes:

r:- type of all the program structures but expressions
-synthesized of **P,C,Cs,A,W**

type:- type of expression
-synthesized of **E and E'**

in: -type expression
-inherited of **E'**

It is extending the first grammar in its last variant discussed in variant