Formal Languages and Compilers

Syntax directed translation

Syntax directed translation

- Basic idea
- Associating information to a language statement
- Equivalently.. Adding attributes to the grammar symbols that represent such statement (context free grammar)
- Is a good approach for obtaining control over the source code of our program

Syntax directed translation

- A syntax directed translation allow us to specify values associated to each production
- This happens using semantic rules
- One or more production can have associated one or more semantic rules

Production	Semantic rules
E → E1 + T	E.code = E1.code || T.code || '+'

Example

- The previous production had 2 non-terminal symbols
- As usual we mark the RH with a number to avoid confusion
- E has an attribute called “code” which is a intuitively a string as well as T
- The semantic rule specify that we obtain E.code by chaining E1.code and T.code

Translation schemas

- A generic translation schema is a notation used to describe the translating process by enriching a grammar with some notation
- Notation are generally (pseudo)code fragments
- A translation schema is similar to a syntax driven translation but the evaluation order of the semantic rule is *explicitly specified*
- stmt→ while(expr) { expr.type == bool? } stmt {…}

General approach to SDT

- Build a parsing tree
- Visit the tree
- While visiting the tree evaluate attributes associated to each node
- In some cases, if the parser is powerful enough (and well instructed), it can 'skip' the generation of the parsing tree and evaluate directly the values of the attributes (we will see a complex example about this)

Attributes

- A syntax directed definition (SDD) is a context free grammar enriched with **attributes** and **semantic rules**
- Attributes are associated to grammar symbols

  E.*code* code an attribute of E

- Semantic rules are associated with grammar productions

  E: E1 * E2 {E.val = E1.*val* * E2.*val*}

Attributes (Cont)

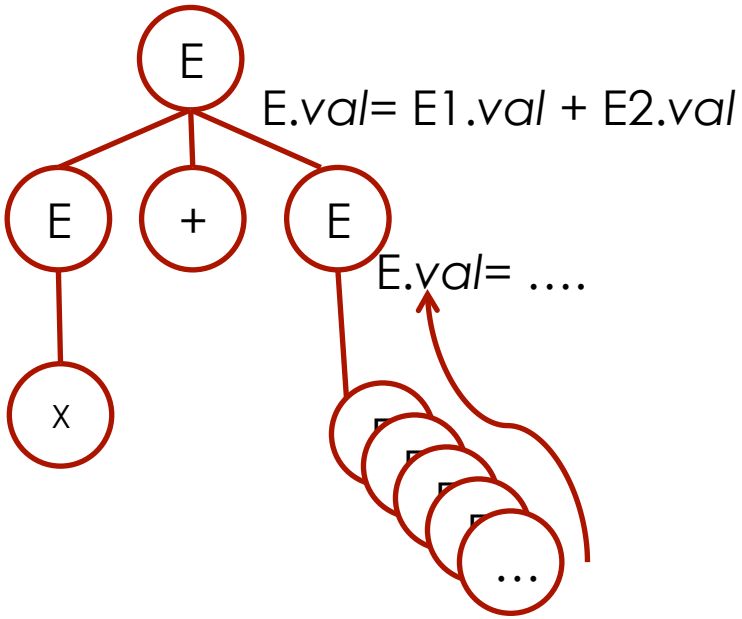
- No fixed type
- An attribute can be whatever you need it to be: numbers, pointers, string, tables, generic types… (NOTE: this is rather theoretical, when you undertake this in practise.. You would like to have some concrete stuff in your hand..)
- For non-terminal symbols we have two kind of attributes:
  - Synthesized Attributes
  - Inherited Attributes

Synthesized Attributes

- Is the attribute of a non-terminal symbol
- An attribute of a non-terminal symbol A, relative to node N of the parsing tree, is such if is defined by a semantic rule associated the production of node N.
- Such production must have A as LH
- L → E \n L.*val*=E.*val*
- "*val*" is the synthesized attribute of L

Synthesized Attributes

- Synthesize attribute is relative to one node N and is uniquely defined upon sons of N and N it self.



Inherited Attributes

- An attribute of a non-terminal symbol B, relative to node N of the parsing tree, is said to be inherited if is defined by the semantic rule associated to the production belonging to the father node of N.

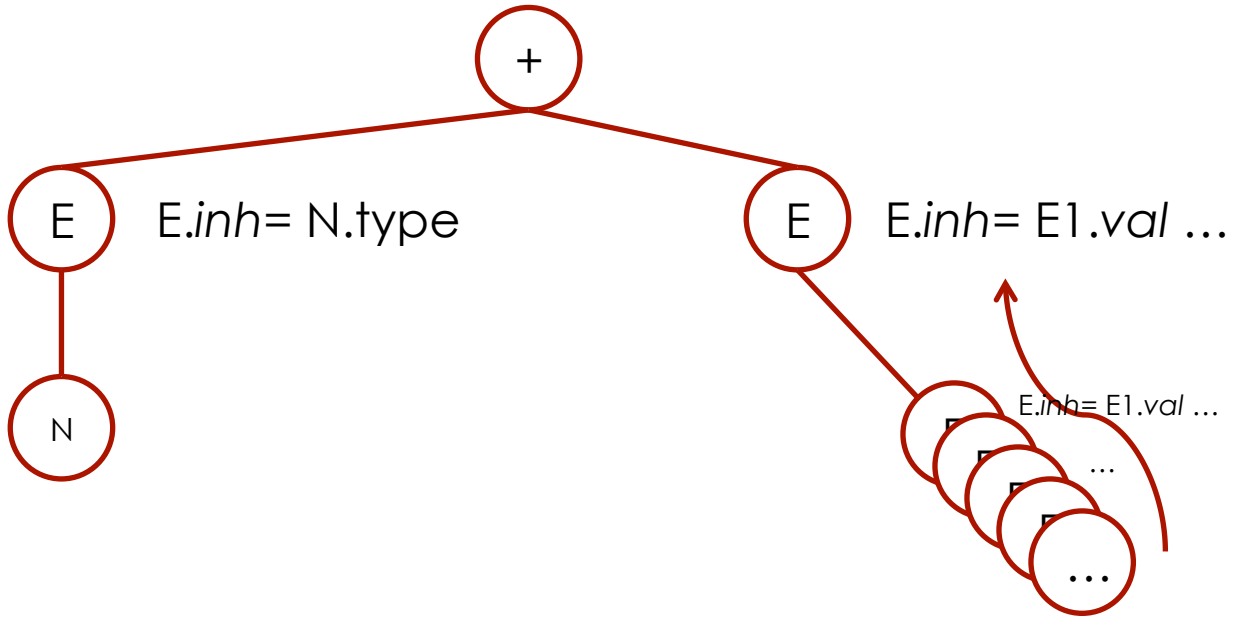
- In this case B must be in the body (RH) of the production.

- T → F T'

  T'.*inh* = F.val
  T.*val* = T'.*syn*

Inherited Attributes

- Inherited attribute is relative to one node N and is uniquely defined upon father of N, N it self and siblings of N.



Example

Production	Semantic Rules
L → E \n	L.val = E.val
E → E + T	E_1.val = E_2.val+ T.val
E → T	E.val = T.val
T → T * F	T_1.val = T_2.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → int	F.val = int.value

Evaluation of attributes

- Is always a good idea to look at the parsing tree to understand the way attributes associated to nodes are evaluated.
- The parser doesn't really need such representation but is more human readable
- Inherited attributes → node N, father of N, siblings of N
- Synthesized attributes → node N, sons of N

Evaluation…example

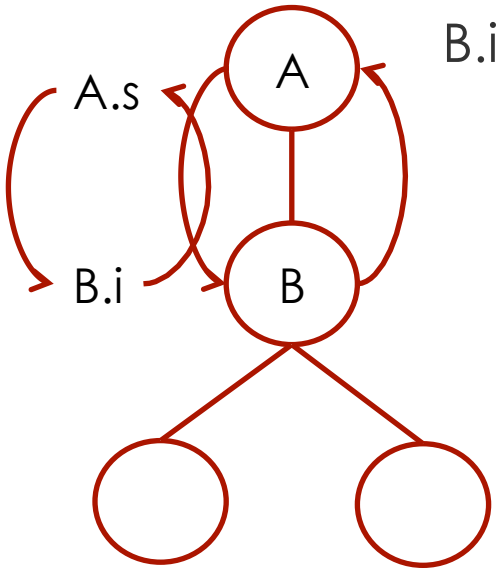
- What does happen if we have

PRODUCTION	SEMANTIC RULES
A→B	A.s = B.i B.i = A.s + 1

Evaluation…example

- What does happen if we have

PRODUCTION	SEMANTIC RULES
A→B	A.s = B.i B.i = A.s + 1

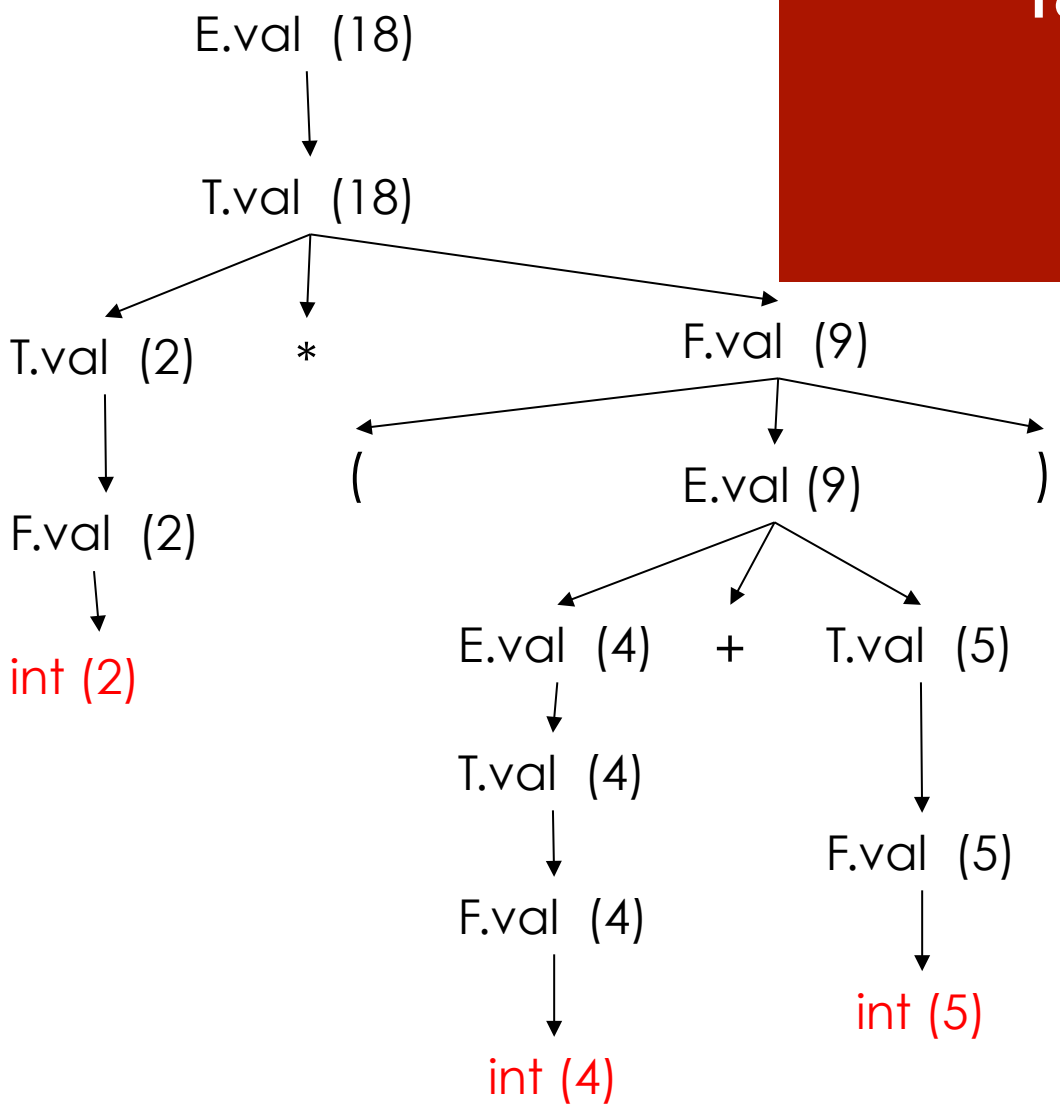


Evaluation of attributes

- A parsing tree that shows the values of the attributes associated to various nodes is called ***annotated syntax tree***
- How do we build such a tree?
- What is the order of evaluation of the attributes?
- Of course before evaluating one attribute of a given node, we must evaluate all its connected attributes

- How do we proceed in the evaluation?

Input : 2 * (4 + 5)



Dependency graph

- Dependency graphs are a useful and powerful tools to shape the evaluation order of each attribute of a node.
- It represents the information flow using attributes of a given parsing tree.
- An attribute *b* **depends** on an attribute *c* if a valid value of *c* must be available in order to find the value of *b*
- The relationship among attributes defines a dependency graph for attribute evaluation

Evaluation orders

- A topological order (topological sort) over the obtained graph give us a valid order that allow us to evaluate the semantic rules
- We can evaluate only acyclic graphs
- If there are no cycles is always possible to find an ordering
- But.. How to grant every time that there are no cycles???

SDD classes

- Is very difficult to grant if there is at least one parsing tree (that serves our purposes) that has no cycles
- To implement our directed translation we can use specific SDD classes
- Such classes can guarantee the existence of an evaluation order
- This is due to the fact that such SDD classes does not admit any dependency graph with cycles

S-attributed definition

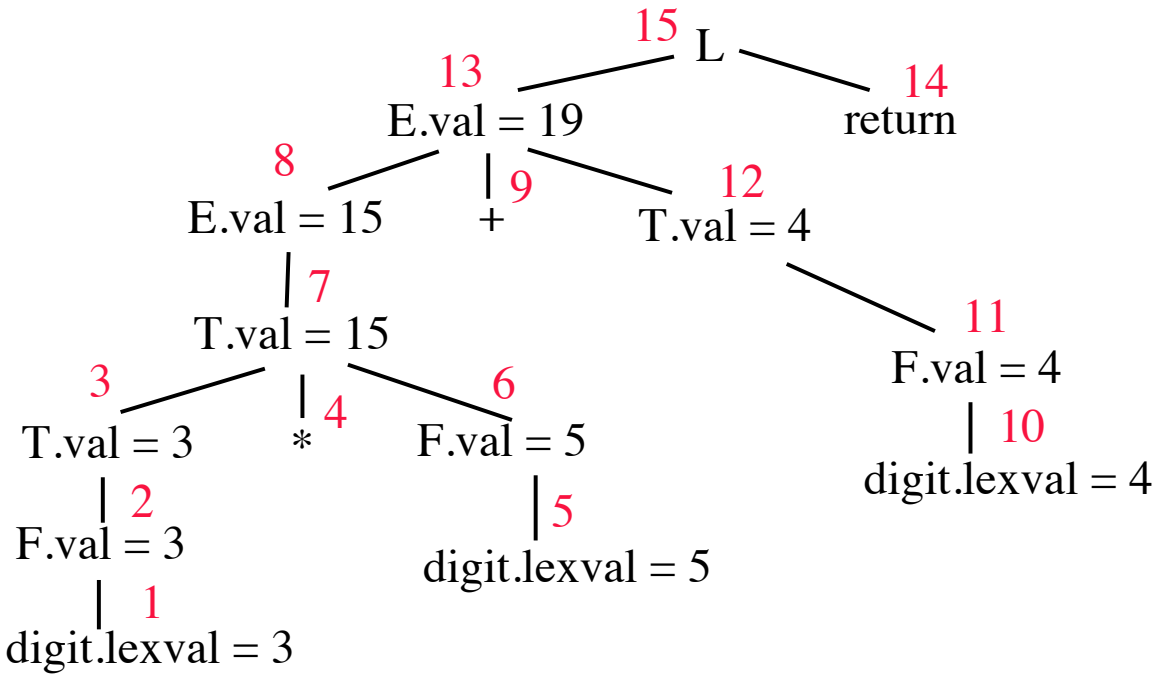
- A **S**yntax **D**irected **D**efinition is said to be ***S-attributed*** iff every and each of its attribute is synthesized

- The previous example is S-attributed

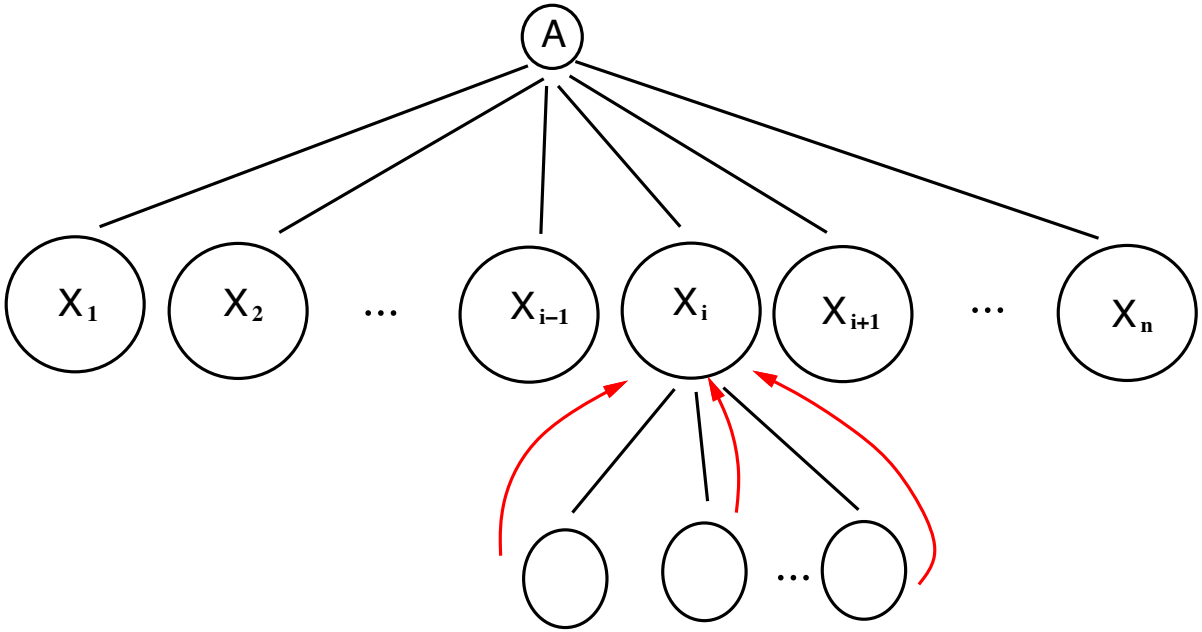
L → E \n	L.val = E.val
E → E + T	E_1.val = E_2.val+ T.val
E → T	E.val = T.val
T → T * F	T_1.val = T_2.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → int	F.val = int.value

- Can be used directly by a LR parser, without building the parsing tree

S-attributed another example



S-attributed another graphical representation



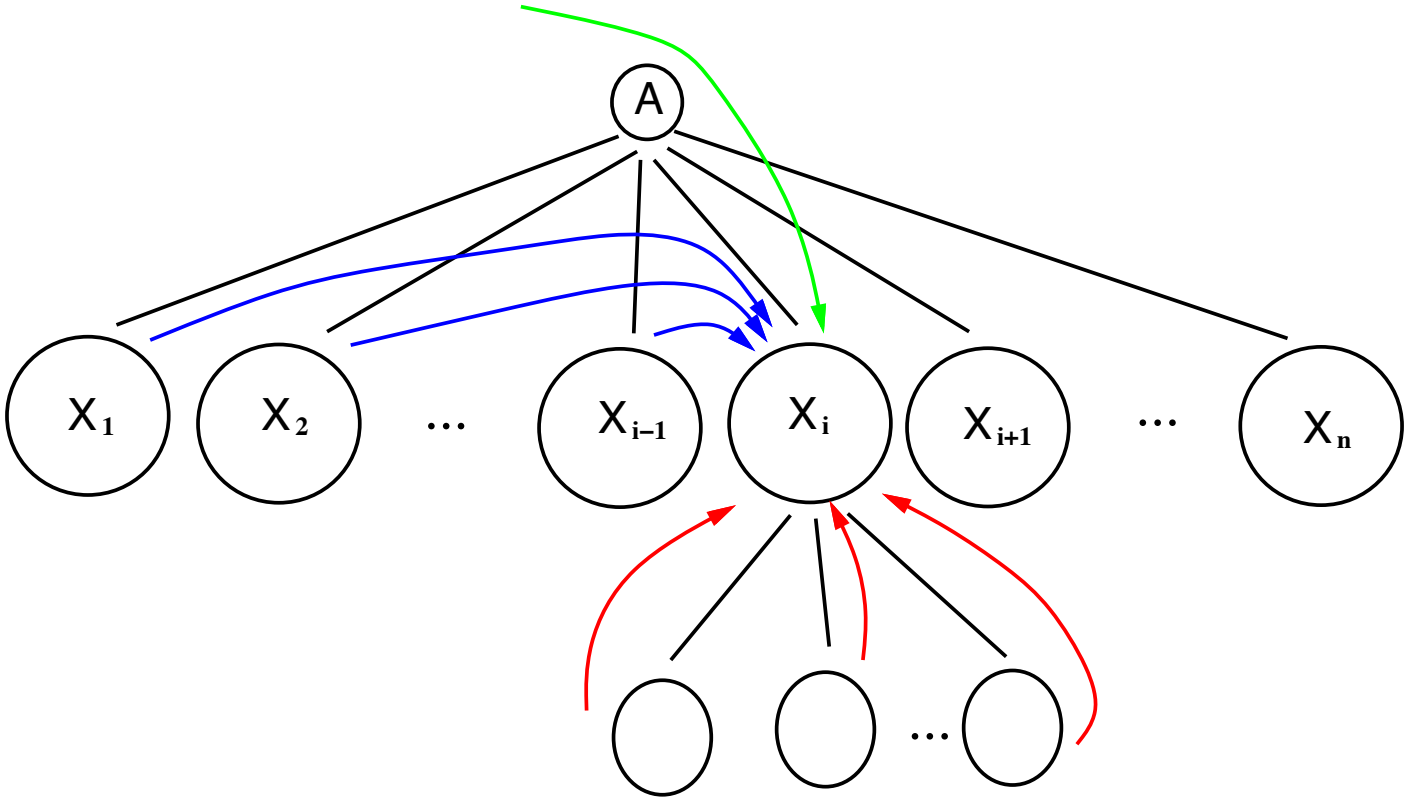
L-attributed definition

- This class has a more strict definition
- L stands for *left* and identifies the property of the class
- Each grammar symbol can have many attribute, the arch created by the dependency anyhow can be oriented only from left to right and ***NOT*** vice versa.

L-attributed definition

- More precisely, an attribute can be
- A synthesized attribute, or
- Inherited attribute with the following restrictions:
- Let $A \rightarrow X_1 \ldots X_n$ be a production to which is associated a semantic rules that computes the value of and ***inherited*** attribute Xi.a. Such rules can only use
  - Inherited attributes associated with the driven of A
  - Attributes, inherited or synthesized, associated to instances of $X_1 \ldots X_{i-1}$ which appear on the left of X_i
  - Any other attribute associated to X_i such that it does not create any circular dependency for X_i

L-attributed graphical representation



L-attributed example

- This example shows an L-attributed grammar

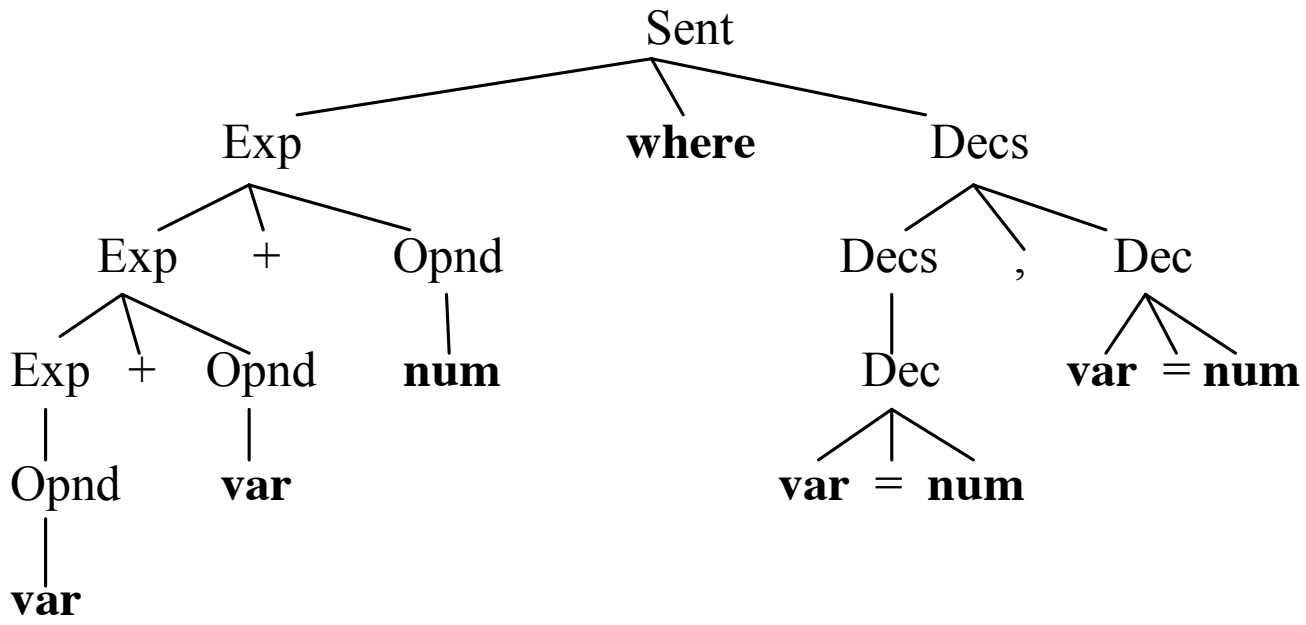
$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *\ F\ T'^{1}$	$T'^{1}.inh = T'.inh * F.val$ $T'.syn = T'^{1}.syn$
$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
$F \rightarrow digit$	$F.val = digit.lexval$

Evaluation of L-attributed

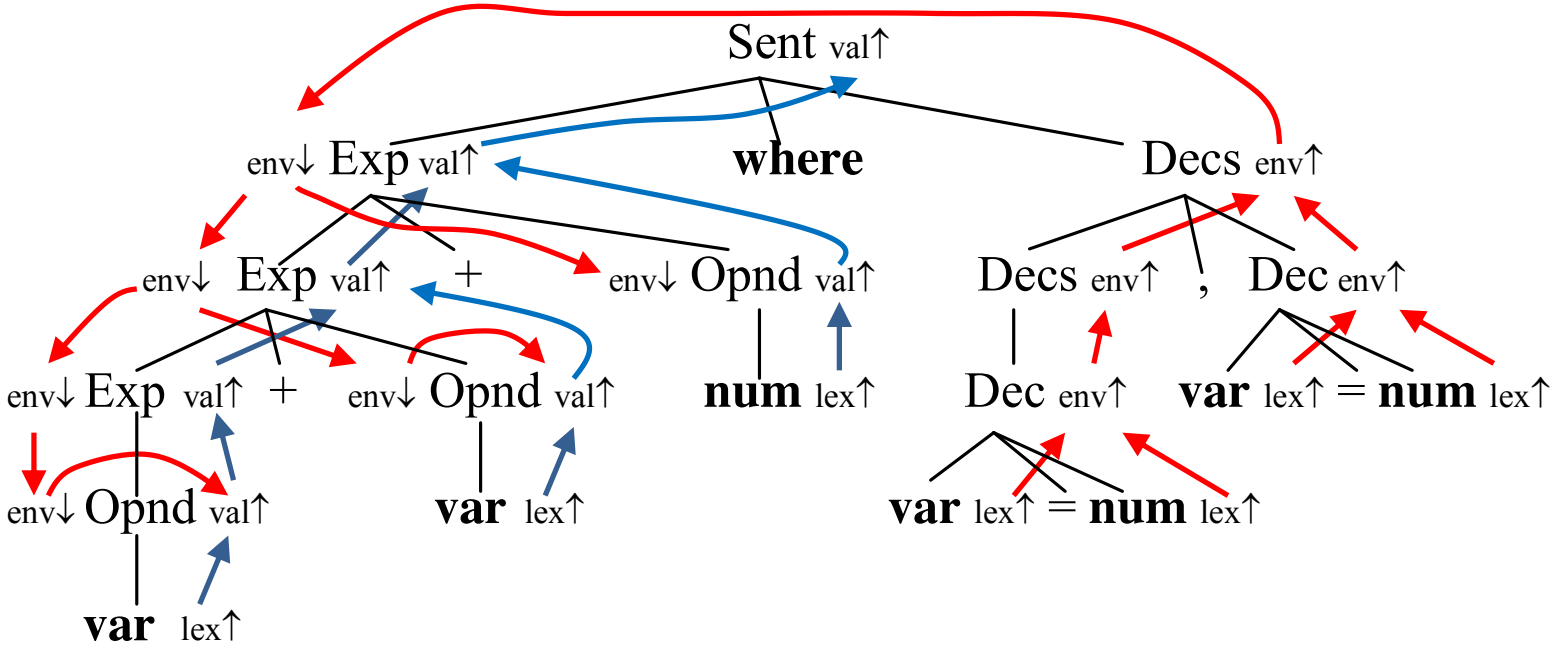
- Build an annotated parsing tree
- Using a LL grammar, with semantic rules.
  In this case we can execute the "translation" directly without building the parsing tree.
- We need some add on in the stack, in order to manage some extra values

A more complex example

- x+y+5 **where** x=5,y=6



A more complex example



A concrete example

- We have already seen that Yacc can manage certain situation

Line : expr		printf($1)	
Expr :	expr '+'term	$$ = $1+ $3	E.val = E_1.val + T.val
	| term	$$ = $1	E.val = T.val
	;		
Term :	term '*' factor	$$ = $1 * $3	T.val = T_1.val * F.val
	| factor		T.val = F.val
	;		
factor:	'('expr')'	$$ = $2	F.val = E.val
	| NUMBER	$$ = $1	F.val = NUMBER.value
	;		

Yacc managed values

- Yacc treats Integers… we have noticed it
- How do we instruct Yacc to manage different type of values….?
- (hint: we saw it already to many times)

Yacc managed values

- Yacc treats Integers… we have noticed it
- How do we instruct Yacc to manage different type of values….?
- We use ***token declaration*** for tokens
  %token <iValue> INTEGER…
  %token <charIndex> IDENTIFIER…
- We specify ***type*** for the drive of the production
  %type <linkedList> stmtList
  %type <nodePointer> expr

Yacc and SDD classes

- Can hold S-attributed grammars…
  use the top of the stack to pass the values of the synthesized attributes
- Can be used with L-attributed grammars..
  Can use global variables to keep track of the values of its elder siblings… BUT…
- Unfortunately, due to its architecture it ***can not*** process inherited values from its parent
  (intuitive hints at slide 51/52)

Semantic rules and actions

- Up to know…
- Lexer gives us the tokens
- Parser gives structure to the syntax of our program
- So.. Syntax is the form of our program…
- If so..what does correspond to semantic rules?

Semantic rules and actions

- Up to know…
- Lexer gives us the tokens
- Parser gives structure to the syntax of our program
- So.. Syntax is the form of our program…
- Semantic rules give meaning to our "*constructs*".. (so are –partially - bound to the actions we specify..)

Semantic rules

- Used to describe how to compute value of attributes in the tree
  (possibly by using other attributes in the tree)
- So far we used semantic rules without really knowing them
- We have used rules by writing action bound to each body of the production
- **NOTA BENE:** so far, nodes may have multiple attributes, they can be grouped in a struct/ dictionary/hash..
  Attributes concern data types, numeric values, simple code fragment, memory addresses..

An example

Production

Stat → **while** (C) $Stat_1$

SDD associated with the previous while

Stat → **while** (C) $Stat_1$

L_1 = new Label();
L_2 = new Label();
$Stat_1$.next = L_1;
C.false = Stat.next;
C.true = L_2;
Stat.code = **label** || L_1 ||C.code|| **label** || L_2 || $Stat_1$.code

We already did something similar.. Recall 2.2 – calc , the compiler version?

SDT syntax directed translation scheme

- Are slightly different from a SDD
- Syntax directed translation schema is a context free grammar enriched (in its various bodies) with pieces of code
- SDT are built directly during the parsing action and does not require effectively the use of a parsing tree.
- *Semantic actions are embedded everywhere in the body of the productions and can execute arbitrary computation*
- They are *executed* left to right (as they can be found somewhere in the middle of the body - MRA)

Mid Rule Actions

- MRA are rules that can interleave the symbols found in the body of a production
- Can refer to components preceding it by using the known notation with ***$n***..
- Mid rule action counts as one component of the production (so you have to count actions and symbols for indexing them correctly)
- Assignment of $$ in a MRA is useless cause it does not take effect
- They are executed before the parser recognizes the remaining part of the production

MRA in literature

- In literature the MRA are know as **markers**
- Marker is an extra symbol which represents the drive of and empty production
  $M \rightarrow \varepsilon$
- We associate to such marker the action we want to execute and we place such marker in the body of the interested production body
- Example:
  $A \rightarrow \{B.i = f(A.i);\}\ B\ C$
  with a marker becomes
  $A \rightarrow M\ B\ C$
  $M \rightarrow \varepsilon$ $\quad\quad\quad\quad\quad\quad$ $\{M.i = A.i;\ M.s = f(M.i);\}$

SDT and SDD classes

- Winning combination for our purposes:
  1) grammar that be recognized by a LR parser
  2) SDD is S-attributed
  note: we already know we can play a bit also with L-attributes, in a restricted way anyhow
- Postfix SDT → when the grammar can be parsed bottom up and is S-attributed
- We just place semantic rules at the end of the body, so that actions are executed upon reducing the production (see S-attributed definition slide 22)
- They are based on the parser stack

SDD vs SDT

- SDD comes in a declarative flavour
  - The author of the grammar specifies what kind of calculation have to be done
  - Evaluation order is not specified – this aspect is left to the parser generator
  - The grammar is human readable
- SDT have a more procedural flavour (imperative style)
  - The author of the grammar specifies both
    - What kind of calculation must be done
    - When calculation have to be done
  - The grammar lose readability but is far efficiently parsable

SDT – while example

- Let's do a step back.
- Recall the previous while example
  Stat → **while** (C) $Stat_1$
  SDT associated with the previous while

Stat → **while** (	{L_1 = new Label();L_2 = new Label();C.false = Stat.next;C.true = L_2;}
C)	{$Stat_1$.next = L_1}
$Stat_1$	{Stat.code = **label** || L_1 ||C.code|| **label** || L_2 || $Stat_1$.code}

SDT – while example (cont)

- The previous SDT can be transformed to work with a LR parser (the grammar is actually LL)
- We add some markers

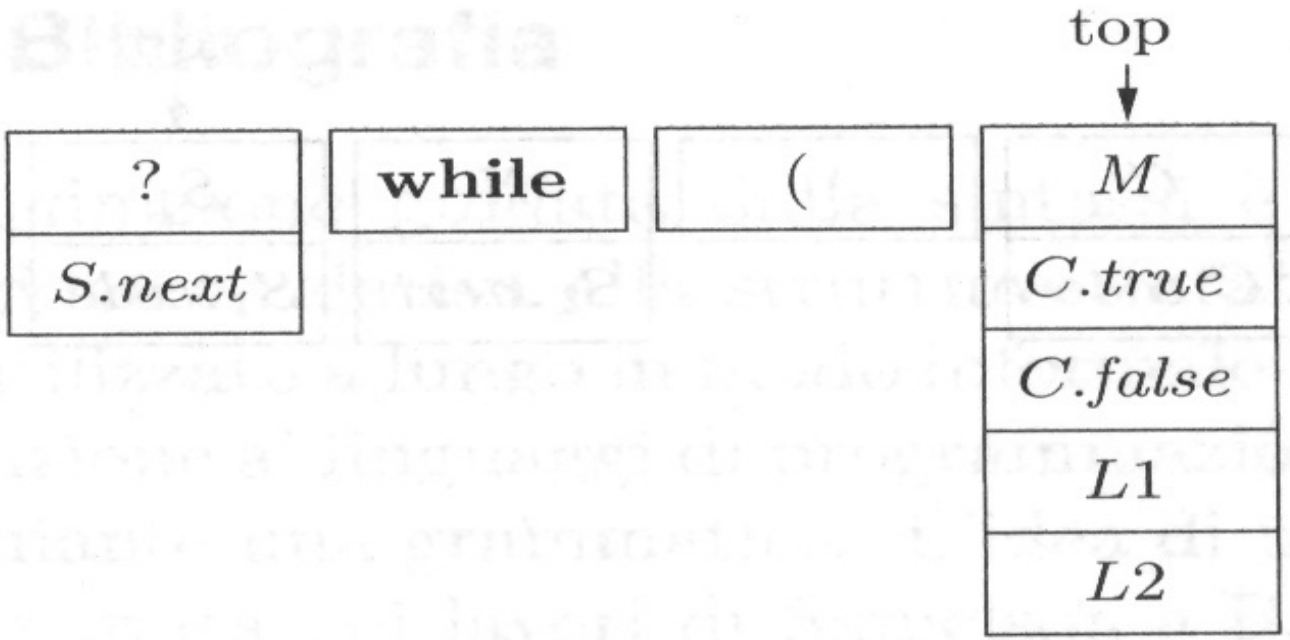
S→ **while** (M C) N S_1 {Stat.code = **label** || L_1 || C.code || **label** || L_2 || $Stat_1$.code}

M→ ε {L_1 = new Label();L_2 = new Label(); C.false = Stat.next;C.true = L_2;}

N → ε {$Stat_1$.next = L_1}

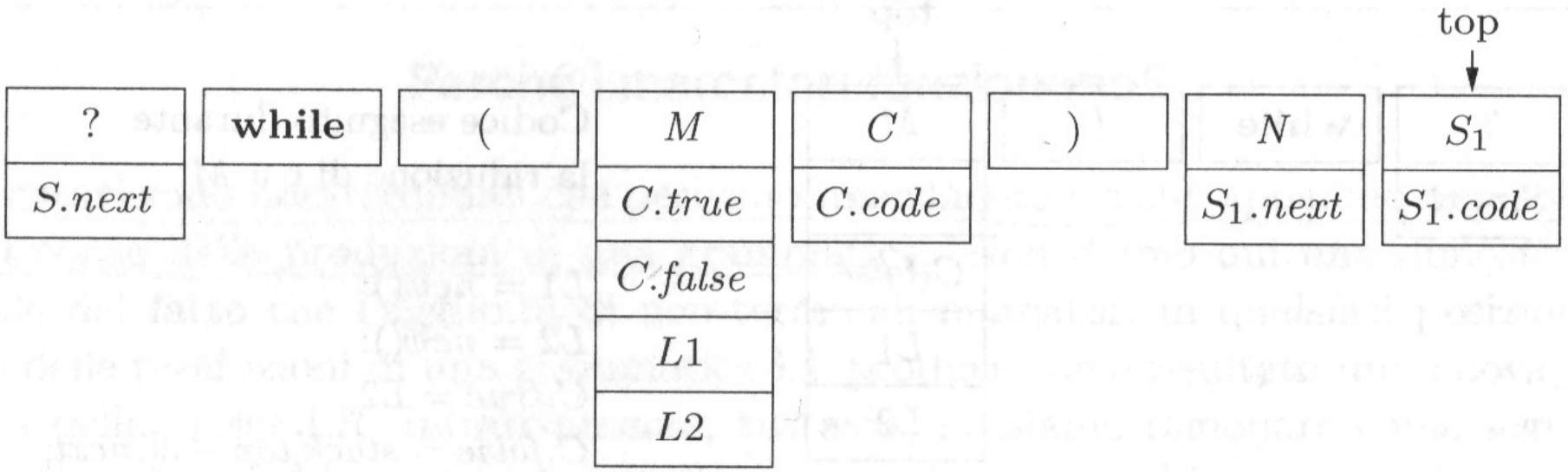
SDT – while example (cont)

- After reducing M



SDT – while example (cont)

- After N and going on



Do markers really work?

- Short answer: **YES**
- Markers are non terminal symbols that derives uniquely ε *and such that they appear only one time in the production of the grammar*
- By adding markers to a LL grammar we can obtain a LR grammar
- Intuitively..
- The grammar is LL and if we have an input string *w* it derives by a non-terminal A by A→ α (This by analysing the first symbol of *w*, or the next if the first is epsilon)
- If we proceed bottom up instead the prefix of *w* can be reduced to α (and consecutively to A) only when the prefix of *w* appears as input (is read).
- By this, if we place markers in any arbitrary position, the parser will take care of the fact that a marker must appear in a given position and that ε must be reduced to such marker while reading the input sequence..

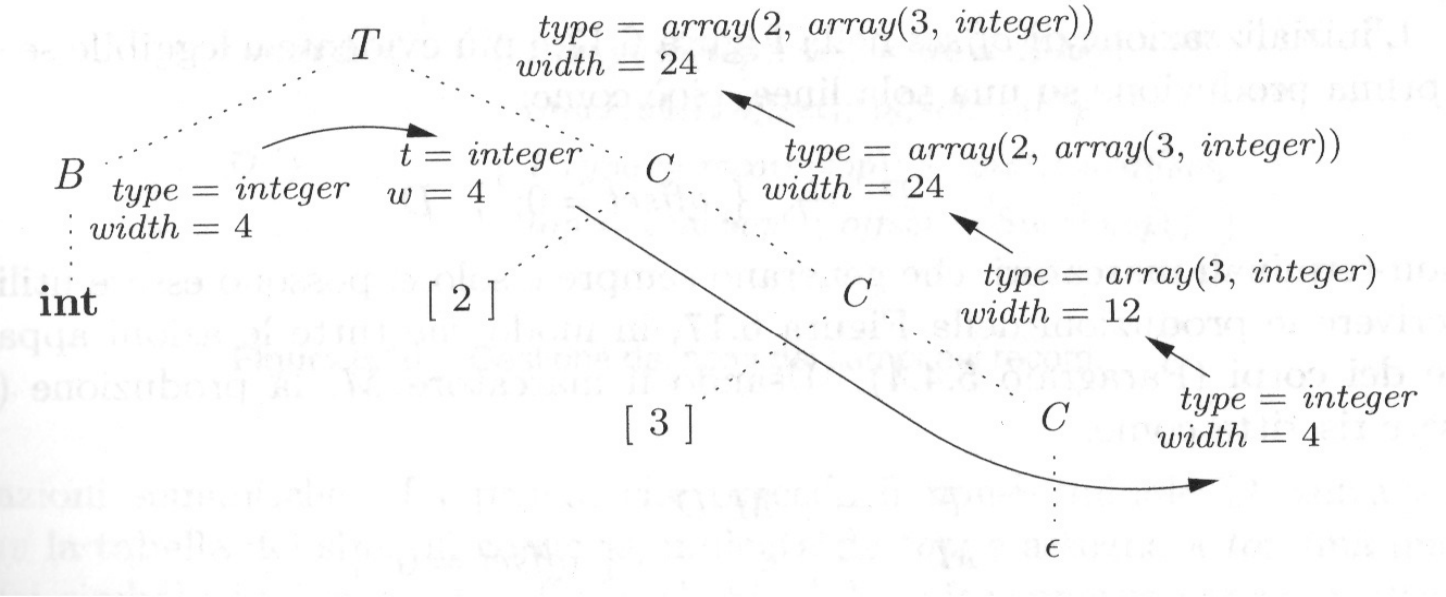
What about SDD L-attributed with LR grammars?

- Short answer : **NO**
- Intuitive answer:
  Say we have a production A → B C in a LR grammar.
  Suppose also that there is an inherited attribute B.i which depends on the inherited attributes of A…
  When the parser reduces B it does not have processed the input part relative to C.. So we are not sure that we are processing the whole body of A→ B C .. This means we can not process B.i and we can not use the semantic rules associated to this production..

Declaration sequences

- We will now proceed by analysing another concrete example → time to code
- Grammar snippet is
  D → T id; D | ε ;
  T → B C;
  T → record { D };
  B → int;
  B → float;
  C → ε ;
  C → [num] C_1;

Declaration sequences (cont)



- We need a way to carry B synthesized value.. Down in the parsing tree… how?

Declaration sequences (cont)

T → B C ;	{t = B.*type*; *w* = B.*width*;} {T.*type* = C.*type*; T.*width* = C.*width*;}
B → int;	{B.*type* = integer; B.*width* = 4;}
B → float;	{B.*type* = float; B.*width* = 8;}
C → ε ;	{C.*type* = t; C.*width* = *w*;}
C → [num] C_1;	{array(num.*value*, C_1.type); C.*width* = num.value * C_1.*width*; }

Accessing variables

- We have now our sequence of declaration..
- We want to access this values now..
  Int i..
  Print (i)
  struct {int portalhold} agent;
  print (agent.portalhold)
- We want to assign values to such variables…
  i = 3;
  i = i + 9 * 7;
  agent.portalhold = agent.portalhold + 1;
- We define a new command in our command production…
  L = R…
- Let's see the code

Bibliography

- http://www-inst.eecs.berkeley.edu/~cs164/fa06/lectures/lecture14.pdf
- http://www.iis.sinica.edu.tw/~tshsu/compiler2013/slides/slide4.pdf
- http://borame.cs.pusan.ac.kr/ai_home/lecture/compiler/2013/CS540-2-lecture6.pdf
- http://www.csi.ucd.ie/staff/acater/2010/30330_L12.pdf
- Compilers 2nd edition – Aho, Lam, Sethi, Ullman
- Bison manual