(01JEUHT) Formal Languages and Compilers

Laboratory N°5

Stefano Scanzio
mail: stefano.scanzio@polito.it
Web: http://www.skenz.it/compilers

Inherited attributes

- Are useful to express the dependency of a production on its context.
- Example:

a , b : int ;

D $\rightarrow$ L ':' T ';'	`L.type = T.type`
L $\rightarrow$ L_1 ',' id	`L1.type = L.type;    put(id.name,L.type)`
L $\rightarrow$ id	`put(id.name,L.type)`
T $\rightarrow$'integer'	`T.type = type_int`

L-attributed grammar

- The order in which attributes are evaluated depends on the order in which the parse tree is created or visited.
- Usually, parser follow the same order of the depth-first visit algorithm.
- An *L*-attributed grammar is defined as a grammar whose attributes' values can be calculated by means of a depth-first visit of the parse tree.
- In these grammars, information propagates from left to right (within the parse tree).
- The previous grammar is not an *L-attributed* grammar
  - Information propagates from right to left
  - CUP manages only L-attributed grammar

L-attributed grammar

- int a, b;

Production	Semantic rules
$D \rightarrow T\ L$ ';'	`L.type = T.type`
$L \rightarrow L_1$ ',' id	`L1.type = L.type` `put(id.name,L.type)`
$L \rightarrow$ id	`put(id.name,L.type)`
$T \rightarrow$ 'integer'	`T.type = type_int`

Calculating inherited attributes

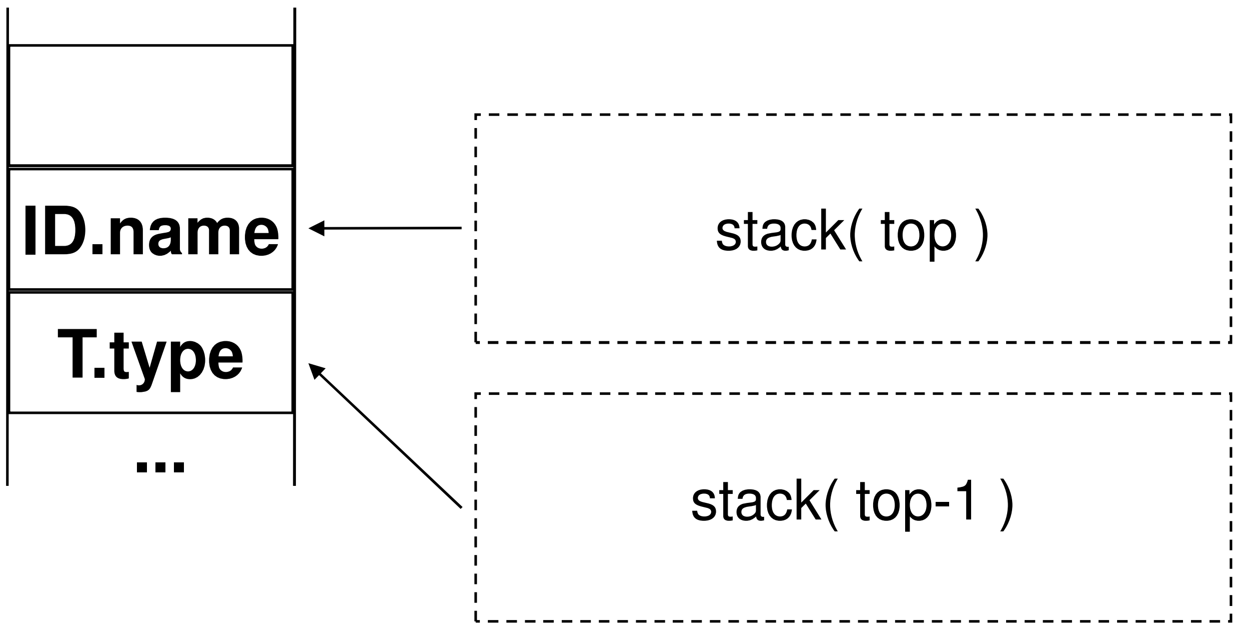
- In a bottom-up parser, memory is not allocated in the semantic stack until the corresponding symbol is recognized.
- This is troublesome for handling inherited attributes.
- If the grammar is an L-attributed one, this issue can be tackled, possibly with the use of markers:
  - Marker: non-terminal that is expanded with ε symbol.

Calculating inherited attributes

- A production with inherited attributes:

D → T lid S	`lid.type = T.type`
lid → ID	`put(ID.name,lid.type)`

Stack before lid is reduced



Calculating inherited attributes (I)

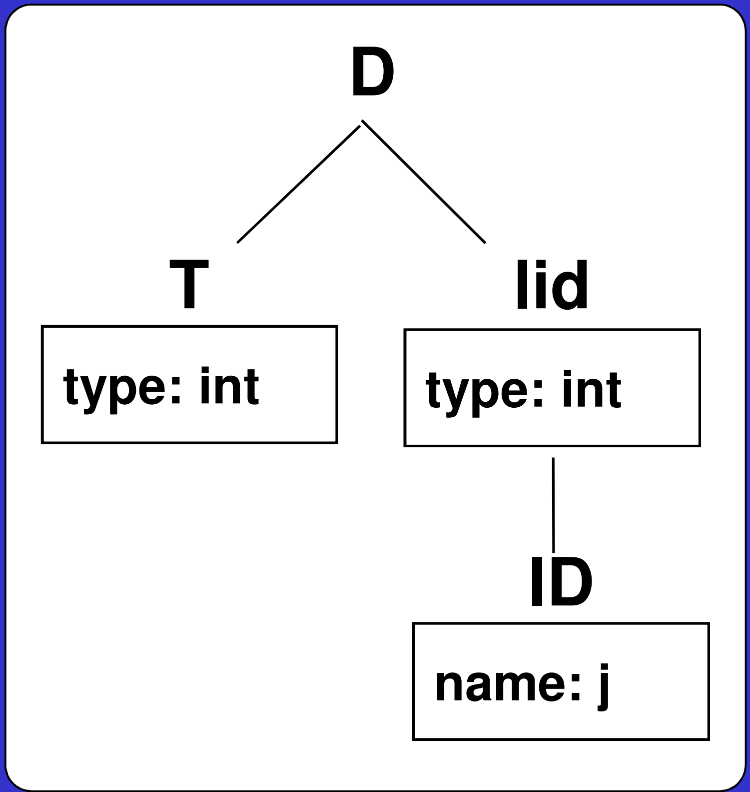
- To access to the semantic values stored in the stack in a given position, use the function:

Object stack(int position)

```
parser code {:
…………………………..
public Object stack ( int position){
   // returns the object at the specified  position
   // from the top (tos) of the stack
      return(((Symbol)stack.
      elementAt(tos+position)).value);
}
…………………………..
:}
```

- *stack(0)* is the semantic value associated with the symbol in the top of the stack;
- *stack(n) is the semantic value associated with the symbol in the position* top+n of the stack

Calculating inherited attributes (II)

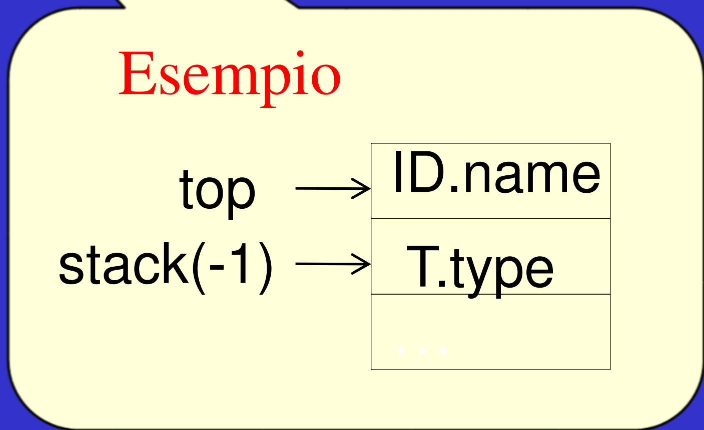


- The 'type' attribute of 'lid' is inherited.
- Its value is present in the semantic stack (in the position of 'T') before 'lid' is created.
- However, it is beyond the semantic scope of the 'lid' production.

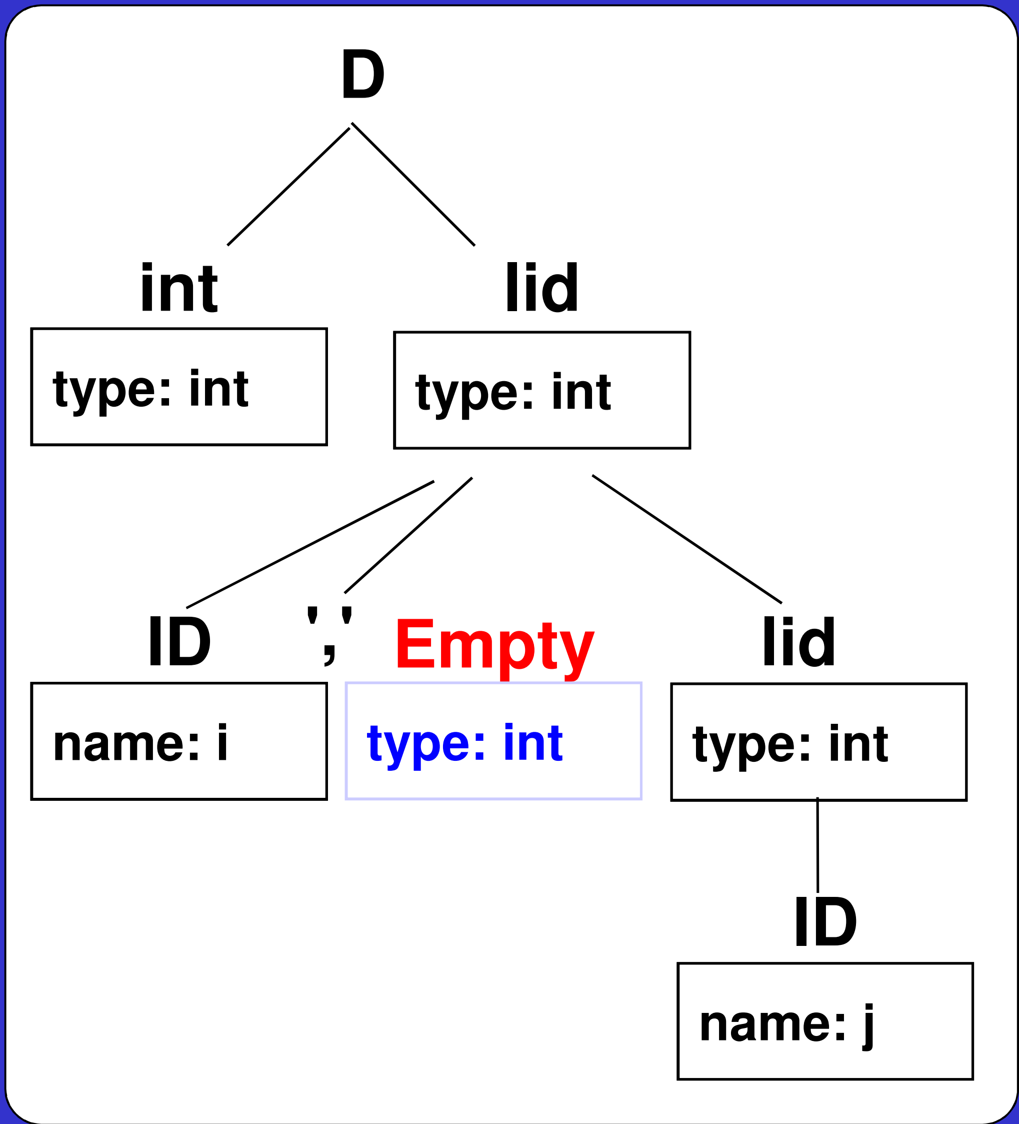
Calculating inherited attributes (III)

With the assumption that the 'lid' symbol is always preceded by a type identifier:

```
lid ::= ID:name {:
        String type = (String) parser.stack(-1);
        RESULT = new String (type);
        put(name, RESULT);
:} ;
```



Calculating inherited attributes by means of markers

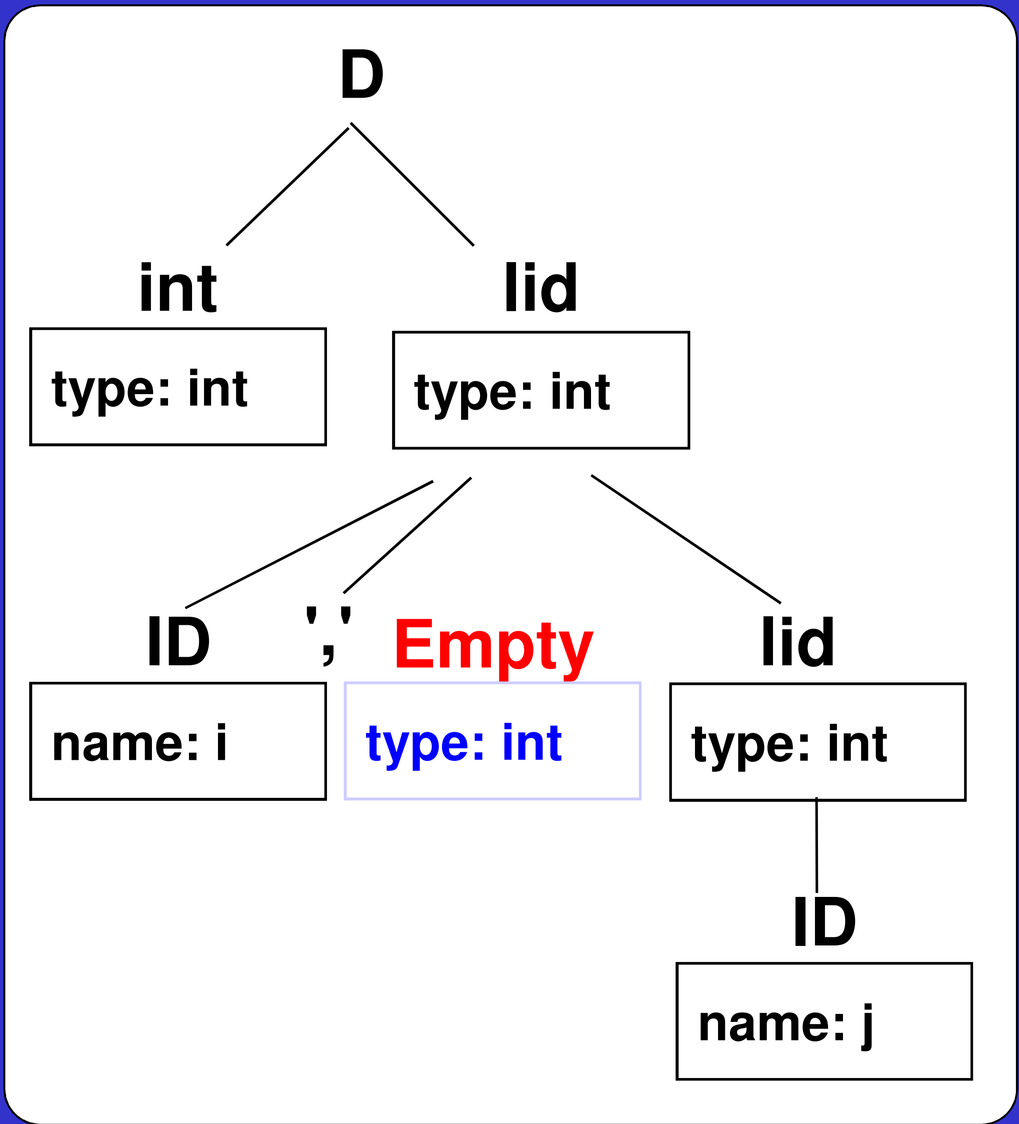


- If the rule lid ::= ID CM lid ; is added, it is not true anymore that 'lid' is always preceded by a type identifier.
- In the case of the rule:

  lid ::= ID;

  the symbol preceding 'ID' in the stack before reducing is 'CM'

Calculating inherited attributes by means of markers



- By adding an empty rule (marker), one can ensure that the rule lid::=ID is preceded by a type semantic value
  - The marker is used to move a semantic value in a desired position in the stack
- IMP: to have easier sematic actions is always better to have left recursive lists
  - lid ::= lid CM ID | ID ;
  - Anyhow, in some grammars, also using left recursive lists, marker are needed

Example:
Calculating inherited attributes by means of markers

```
lid ::= ID:name {:
        RESULT = (String) parser.stack(-1);
        put(name, RESULT);
:} ;

lid ::= ID:name CM Empty lid {:
        RESULT = (String) parser.stack(-1);
        put(name, RESULT);
:} ;

Empty ::= {:
        RESULT = (String) parser.stack(-2);
:} ;
```

GRAMMAR

D ::= T lid S ;
Lid ::= ID CM Empty lid
| ID ;
Empty ::= /* ε */ ;

Intermediate actions

- In order to avoid explicitly introducing a non-terminal with an empty production, one can use in the right-hand side of the production an **intermediate action**.
- Intermediate actions are automatically substituted with a non-terminal symbol, which in turn is given by an empty production.

Intermediate actions: example

- The following code:

```
lid ::= ID:name CM Empty lid ;
Empty ::= ;
```

- can be rewritten as:

```
lid ::= ID:name CM {:
        RESULT = (String) parser.stack(-2);
    :}
    lid {:
        RESULT = (String) parser.stack(-1);
        put(name, RESULT);
:} ;
```

scanner.jflex

Example: marker (I)

```
import java_cup.runtime.*;
%%
%cup
%unicode

nl     =  \n | \r | \r\n
id     =  [a-zA-Z][a-zA-Z0-9_]*
type        =  int | float | char | double

%%

","                     { return new Symbol(sym.CM);}
";"                     { return new Symbol(sym.S);}

{type}                  { return new Symbol( sym.TYPE, new String(yytext()) ); }

{id}            { return new Symbol(sym.ID, new String(yytext()) ); }

{nl} | " " | \t     { ; }
```

parser.cup

Example: marker (II)

```
import java_cup.runtime.*;

parser code {:
    // Return semantic value of symbol in position (position)
    public Object stack(int position) {
            return (((Symbol)stack.elementAt(tos+position)).value);
    }
:};

terminal CM, S;
terminal String TYPE, ID;
non terminal goal, list_decl;
non terminal String decl, lid;

start with goal;

goal ::= list_decl      {: System.out.println("PARSER: Recognized grammar!!");
:};

list_decl ::= | list_decl decl;
```

parser.cup

Example: marker (III)

```
decl ::= TYPE lid:x S {:
    System.out.println("PARSER: Found declaration of type: " + x);
:};

lid ::= ID:name CM {:
            RESULT = (String) parser.stack(-2);
    :}
    lid {:
            RESULT = (String) parser.stack(-1);
            System.out.println("PARSER: put(" + name + ", " + RESULT + ")");
:} ;


lid ::= ID:name {:
            RESULT = (String) parser.stack(-1);
            System.out.println("PARSER: put(" + name + ", " + RESULT + ")");
:} ;
```

Transforming the grammar

- It is possible to avoid using inherited attributes by transforming the grammar.

D → L ':' T

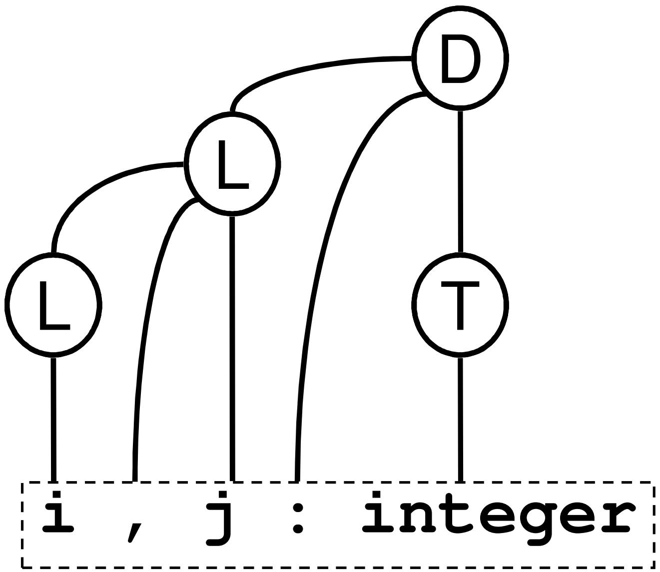
T → **integer** | **real**

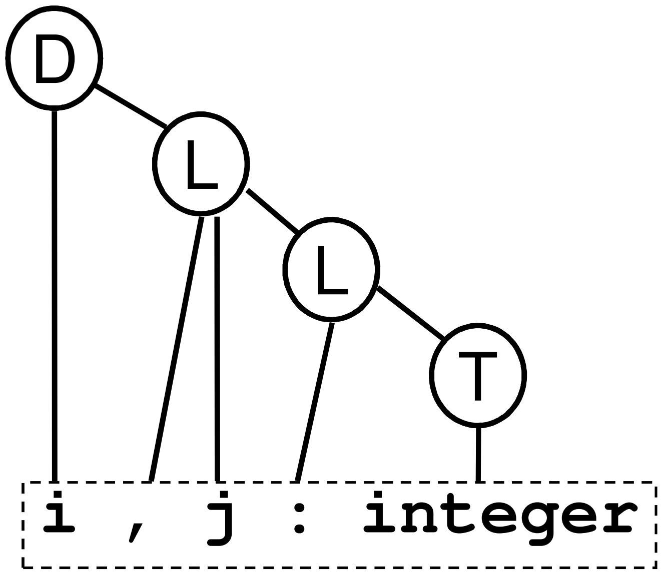
L → L ',' **id** | **id**

D → **id** L

L → ',' **id** L | ':' T

T → **integer** | **real**





Handling semantic errors

- Semantic errors are usually handled in the actions associated to productions
- Usually, actions verify:
  - That operands types are compatible
  - That variables and functions are declared
  - That the parameters passed to a function are coherent with the function prototype

Intermediate code generation: the WHILE statement

- As an example of intermediate code generation, a simple WHILE statement :

```
while_c ::= WHILE ( a > 0) { /* something */ }
                    cond         stmt
```

- can be translated in the following intermediate code:

```
L0:    EVAL cond
       GOTOF L1
       stmt
       GOTO L0
L1:
```

- Where GOTOF is a jump instruction executed only if the result of the above EVAL command is 0 (i.e., FALSE)
- L0 and L1 are labels

Intermediate code generation: the WHILE statement

- A possible solution of the WHILE problem that uses inherited attributes is:

```
wc ::= WHILE cond NT0:x stmt {: Integer[] l = x;
                                System.out.println( "GOTO L" +l[0]);
                                System.out.print( "L"+l[1]+":"); :};
NT0 ::= {: RESULT = new Integer[2];
           RESULT[0] = genLabel(); //L0:
           RESULT[1] = genLabel(); //L1:
           System.out.print( "L"+RESULT[0]+":");
           System.out.println( "EVAL"+parser.stack(0));
           System.out.println( "GOTOF L"+RESULT[1]); :};
```