UNIT – III

SEMANTIC ANALYZER

Static semantics

- Dynamic semantics
- Attribute grammars
- Abstract syntax trees

Syntax concerns the form of a valid program, while *semantics* concerns its meaning

- Context-free grammars are not powerful enough to describe certain rules, e.g. checking variable declaration with variable use
- *Static semantic* rules are enforced by a compiler at compile time
- Implemented in semantic analysis phase of the compiler
- Examples:
- Type checking
- Identifiers are used in appropriate context
- Check subroutine call arguments
- Check labels

Dynamic Semantics

- *Dynamic semantic* rules are enforced by the compiler by generating code to perform the checks at run-time
- Examples:
- Array subscript values are within bounds
- Arithmetic errors
- Pointers are not dereferenced unless pointing to valid object
- A variable is used but hasn't been initialized
- Some languages (Euclid, Eiffel) allow programmers to add explicit dynamic semantic checks in the form of assertions, e.g.

assert denominator not= 0

- When a check fails at run time, an exception is raised

Attribute Grammars

- An attribute grammar "connects" syntax with semantics
- Each grammar production has a *semantic rule* with *actions* (e.g. assignments) to modify values of *attributes* of (non)terminals
- A (non)terminal may have any number of attributes
- Attributes have values that hold information related to the (non)terminal
- General form:

production **semantic rule**

<*A*> ::= <*B*> <*C*> *A*.a := ...; *B*.a := ...; *C*.a := ...

- Semantic rules are used by a compiler to enforce static semantics and/or to produce an abstract syntax tree while parsing tokens
- Can also be used to build simple language interpreters

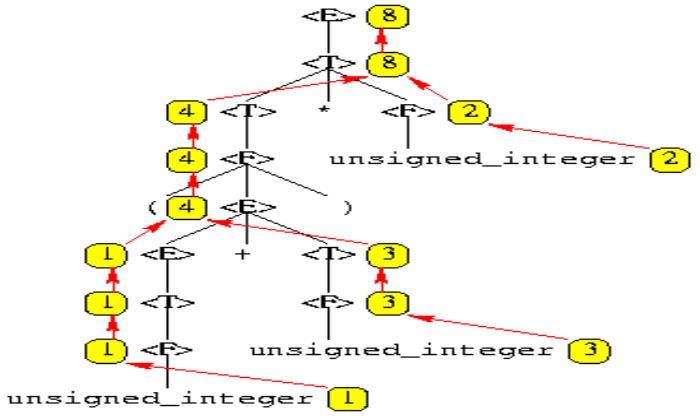
Example Attributed Grammar

- The val attribute of a (non)terminal holds the subtotal value of the subexpression
- Nonterminals are indexed in the attribute grammar to distinguish multiple occurrences of the nonterminal in a production

Production semantic rule

<*E*1> ::= <*E*2> + <*T*>
*E*1.val := *E*2.val + *T*.val
<*E*1> ::= <*E*2> - <*T*>
*E*1.val := *E*2.val - *T*.val
<*E*> ::= <*T*>
E.val := *T*.val
<*T*1> ::= <*T*2> * <*F*>
*T*1.val := *T*2.val * *F*.val
<*T*1> ::= <*T*2> / <*F*>
*T*1.val := *T*2.val / *F*.val
<*T*>::= <*F*>
T.val := *F*.val
<*F*1> ::= - <*F*2>
*F*1.val := -*F*2.val
<*F*>::= (<*E*>)
F.val := *E*.val
<*F*>::= unsigned_int
F.val := unsigned_int.val

Decorated Parse Trees



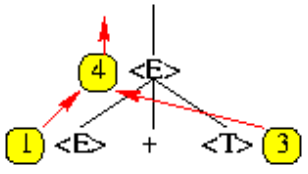
A parser produces a parse tree that is *decorated* with the attribute values
Example decorated parse tree of
(1+3)*2 with the val attributes

Synthesized Attributes

Synthesized attributes of a node hold values that are computed from attribute values of the *child* nodes in the parse tree and therefore information flows upwards

production
semantic rule

<*E*1>::= <*E*2> + <*T*>
*E*1.val := *E*2.val + *T*.val



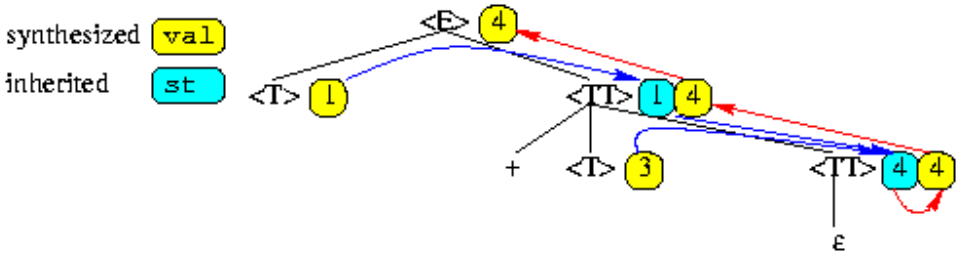
Inherited Attributes

Inherted attributes of *child* nodes are set by the *parent* node and therefore information flows downwards

Production semantic rule

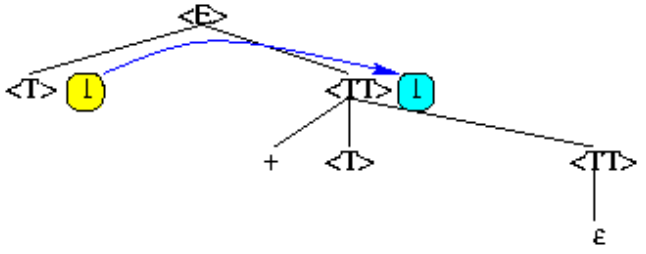
<*E*> ::= <*T*> <*TT*>
TT.st := *T*.val; *E*.val := *TT*.val
<*TT*1> ::= + <*T*> <*TT*2>
*TT*2.st := *TT*1.st + *T*.val; *TT*1.val := *TT*2.val
<*TT*> ::= ε

TT.val := *TT*.st



Attribute Flow

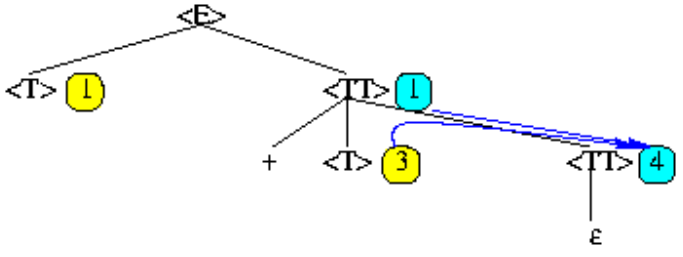
An *attribute flow algorithm* propagates attribute values through the parse tree by traversing the tree according to the *set* (write) and *use* (read) dependencies (an attribute must be set before it is used)



Production semantic rule

<*E*> ::= <*T*> <*TT*>

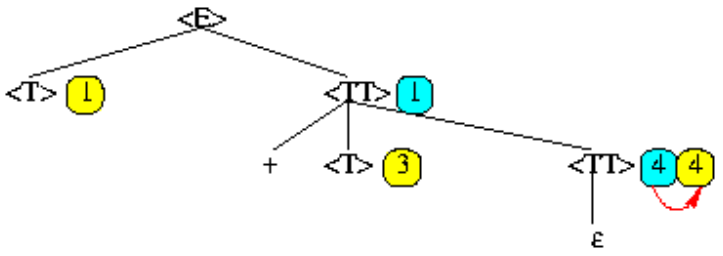
TT.st := *T*.val



Production semantic rule

<*TT1*> ::= + <*T*> <*TT2*>

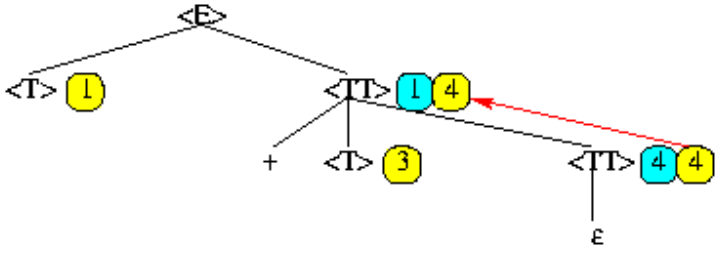
*TT*2.st := *TT*1.st + *T*.val



Production semantic rule

<*TT*> ::= ε

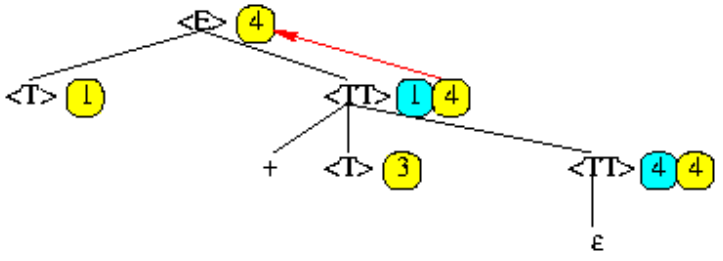
TT.val := *TT*.st



Production semantic rule

<*TT*1> ::= + <*T*> <*TT*2>

*TT*1.val := *TT*2.val



Production semantic rule

<*E*> ::= <*T*> <*TT*>

E.val := *TT*.val

S- and L-Attributed Grammars

- A grammar is called *S-attributed* if all attributes are synthesized
- A grammar is called *L-attributed* if the parse tree traversal to update attribute values is always left-to-right and depth-first
- Synthesized attributes always OK
- Values of inherited attributes must be passed down to children from left to right
- Semantic rules can be applied immediately during parsing and parse trees do not need to be kept in memory
- This is an essential grammar property for a one-pass compiler
- An S-attributed grammar is a special case of an Lattributed Grammar

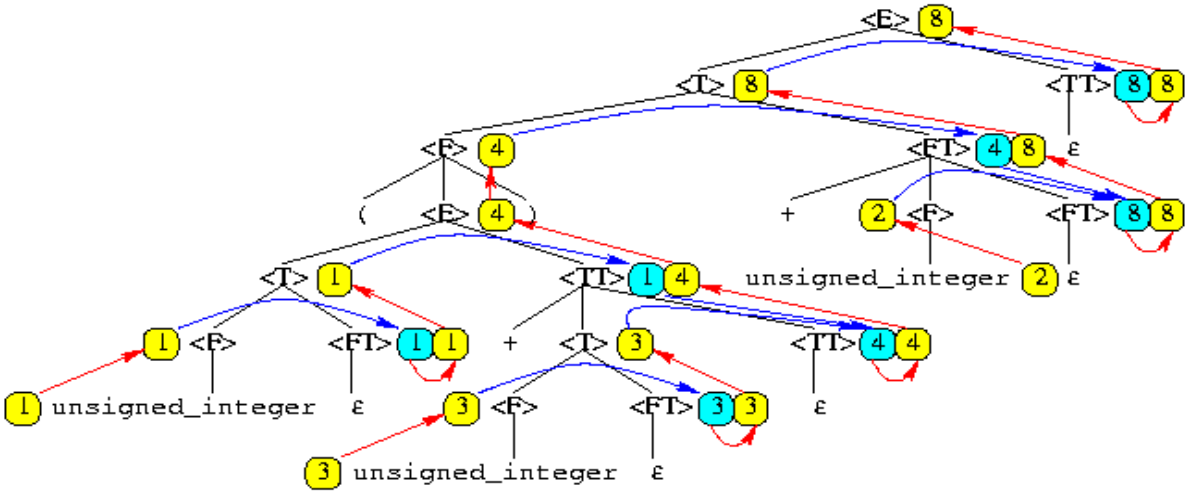
Example L-Attributed Grammar Implements a calculator

Production semantic rule

<*E*>::= <*T*> <*TT*>
<*TT*1> ::= + <*T*> <*TT*2>
<*TT*1>::= - <*T*> <*TT*2>
<*TT*>::= ε
<*T*>::= <*F*> <*FT*>
<*FT*1>::= * <*F*> <*FT*2>
<*FT*1>::= / <*F*> <*FT*2>
<*FT*>::= ε
<*F*1> ::= - <*F*2>
<*F*>::= (<*E*>)
<*F*>::= unsigned_int
TT.st := *T*.val; *E*.val := *TT*.val
*TT*2.st := *TT*1.st + *T*.val; *TT*1.val := *TT*2.val
*TT*2.st := *TT*1.st - *T*.val; *TT*1.val := *TT*2.val
TT.val := *TT*.st
FT.st := *F*.val; *T*.val := *FT*.val
*FT*2.st := *FT*1.st * *F*.val; *FT*1.val := *FT*2.val
*FT*2.st := *FT*1.st / *F*.val; *FT*1.val := *FT*2.val
FT.val := *FT*.st
*F*1.val := -*F*2.val
F.val := *E*.val
F.val := unsigned_int.val

Example Decorated Parse Tree

- Fully decorated parse tree of **(1+3)*2**


<E> 8
<T> 8
<TT> 8 8
<F> 4
<FT> 4 8
ε
(<E> 4)
+ 2 <F>
<FT> 8 8
<T> 1
<TT> 1 4
unsigned_integer 2
ε
1 <F>
<FT> 1 1
+ <T> 3
<TT> 4 4
1 unsigned_integer
ε
3 <F>
<FT> 3 3
ε
3 unsigned_integer
ε