COP4020 Programming Languages

Semantics

Robert van Engelen & Chris Lacher

Overview

- Static semantics

- Dynamic semantics

- Attribute grammars

- Abstract syntax trees

Static Semantics

- *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
$\langle A \rangle ::= \langle B \rangle \langle C \rangle$	$A.a := ...; B.a := ...; C.a := ...$

- 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 distinghuish multiple occurrences of the nonterminal in a production
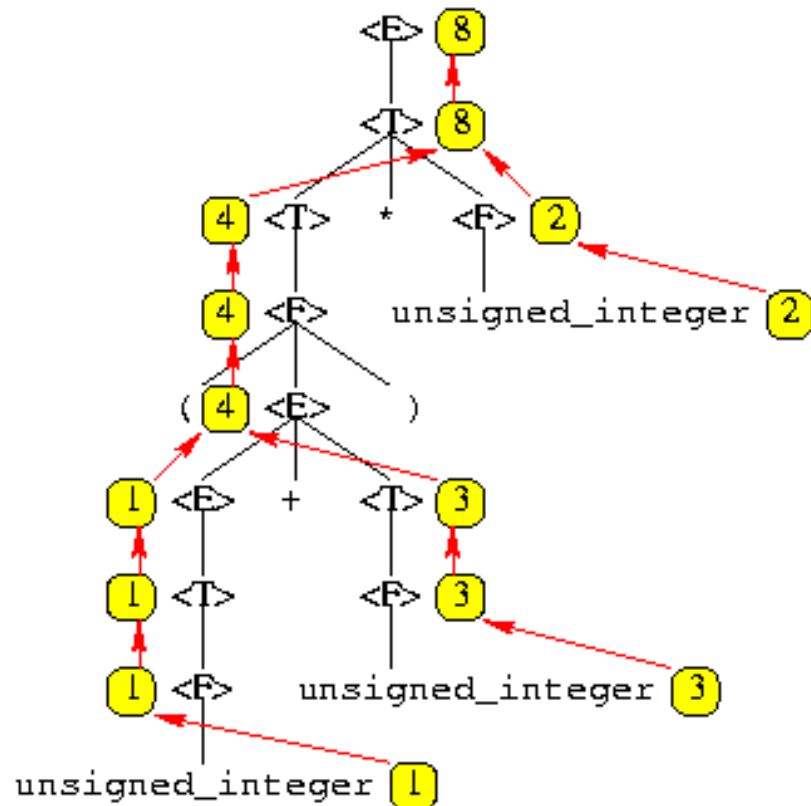
production	semantic rule
$<E_1>$ ::= $<E_2>$ + $<T>$	E_1.val := E_2.val + T.val
$<E_1>$ ::= $<E_2>$ - $<T>$	E_1.val := E_2.val - T.val
$<E>$ ::= $<T>$	E.val := T.val
$<T_1>$ ::= $<T_2>$ * $<F>$	T_1.val := T_2.val * F.val
$<T_1>$ ::= $<T_2>$ / $<F>$	T_1.val := T_2.val / F.val
$<T>$ ::= $<F>$	T.val := 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
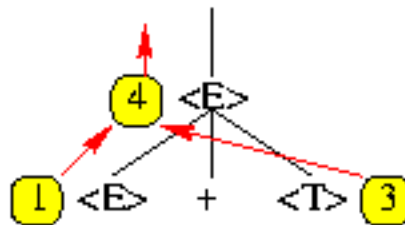
Synthesized Attributes

■ *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
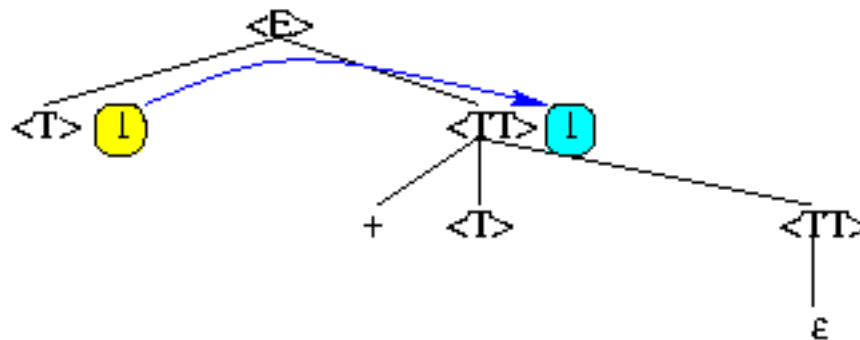
$<TT>$ $::= \varepsilon$ TT.val := TT.st
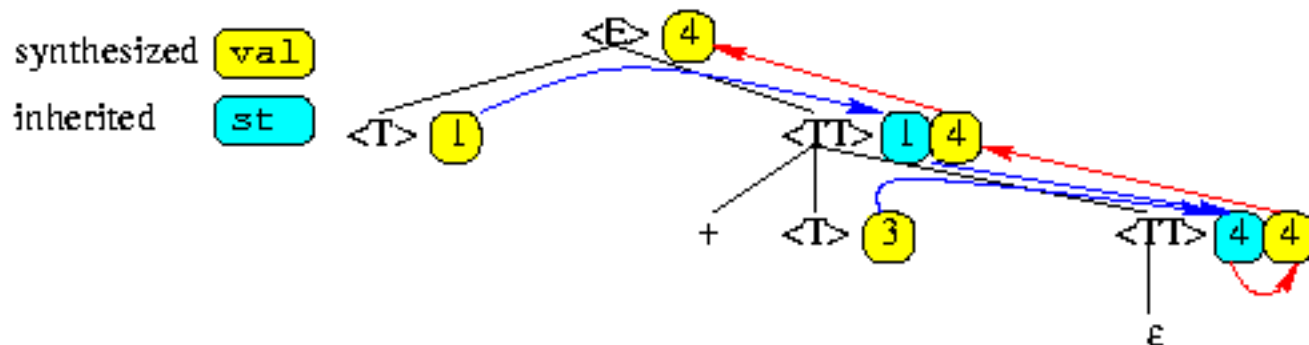
Attribute Flow

- 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

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)
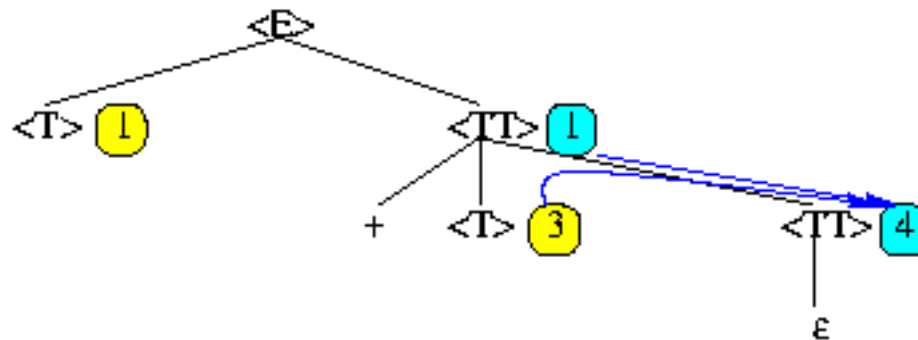

production   **semantic rule**

$<TT_1>$ ::= + $<T>$ $<TT_2>$ TT_2.st := TT_1.st + T.val

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
<*TT*> ::= ε

semantic rule
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)
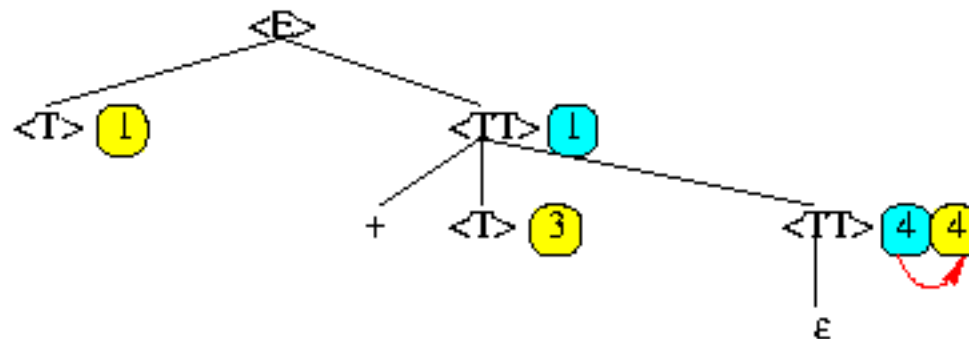

production   **semantic rule**

$<TT_1> ::= + <T> <TT_2>$ $TT_1.\text{val} := TT_2.\text{val}$

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)
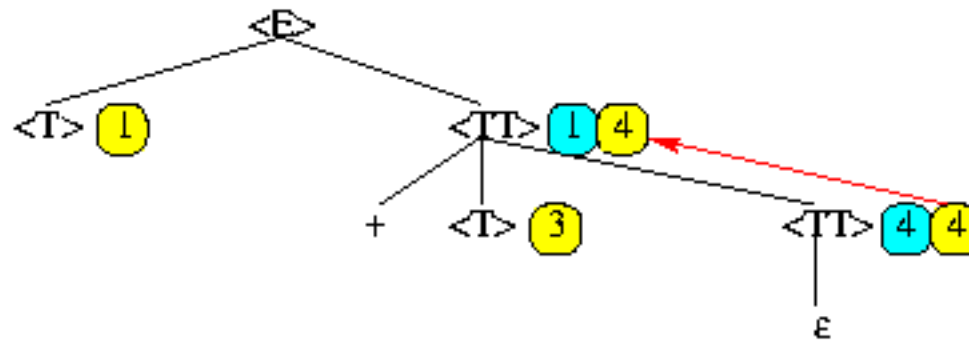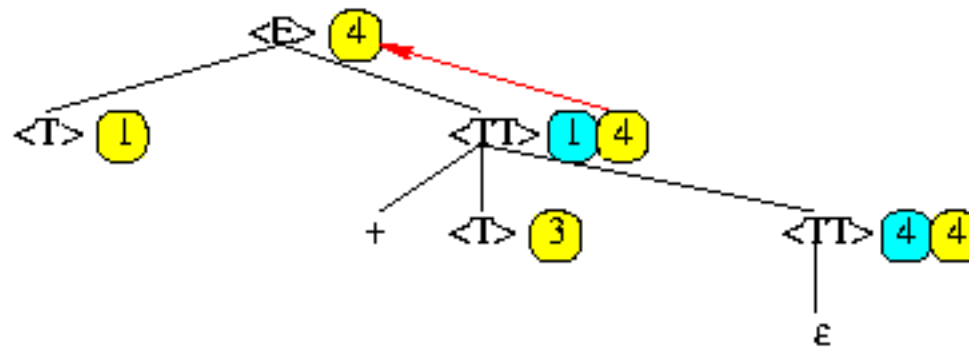

production
<E> ::= <T> <TT>

semantic rule
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 L-attributed grammar

Example L-Attributed Grammar

- Implements a calculator

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
$<TT_1>$ ::= - $<T> <TT_2>$	TT_2.st := TT_1.st - T.val; TT_1.val := TT_2.val
$<TT>$ ::= ε	TT.val := TT.st
$<T>$ ::= $<F> <FT>$	FT.st := F.val; T.val := FT.val
$<FT_1>$ ::= * $<F> <FT_2>$	FT_2.st := FT_1.st * F.val; FT_1.val := FT_2.val
$<FT_1>$ ::= / $<F> <FT_2>$	FT_2.st := FT_1.st / F.val; FT_1.val := FT_2.val
$<FT>$ ::= ε	FT.val := FT.st
$<F_1>$ ::= - $<F_2>$	F_1.val := -F_2.val
$<F>$ ::= ($<E>$)	F.val := E.val
$<F>$ ::= unsigned_int	F.val := unsigned_int.val

Example Decorated Parse Tree

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

Recursive Descent Parsing with L-Attributed Grammars

- Semantic rules are added to the bodies of the recursive descent functions and placed appropriately between the function calls

- Inherited attribute values are input arguments to the functions
  - Argument passing flows downwards in call graphs
- Synthesized attribute values are returned by functions
  - Return values flow upwards in call graphs

Example

production

$<E>$ ::= $<T> <TT>$

$<TT_1>$ ::= + $<T> <TT_2>$

$<TT_1>$ ::= - $<T> <TT_2>$

$<TT>$ ::= ε

semantic rule

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

```
procedure E()
 Tval = T();
 Eval = TT(Tval);
 return Eval;
procedure TT(TTst)
 case (input_token())
 of '+': match('+');
        Tval = T();
        TTval = TT(TTst + Tval);
 of '-': match('-');
        Tval = T();
        TTval = TT(TTst - Tval);
 otherwise: TTval = TTst;
 return TTval;
```

Constructing Abstract Syntax Trees with Attribute Grammars

■ Three operations to create nodes for an AST tree that represents expressions:

   □ *mk_bin_op*(*op*, *left*, *right*): constructs a new node that contains a binary operator *op* and AST sub-trees *left* and *right* representing the operator's operands and returns pointer to the new node

   □ *mk_un_op*(*op*, *node*): constructs a new node that contains a unary operator *op* and sub-tree *node* representing the operator's operand and returns pointer to the new node

   □ *mk_leaf*(*value*): constructs an AST leaf that contains a value and returns pointer to the new node

An L-Attributed Grammar to Construct ASTs

- Semantic rules to build up an AST

production

semantic rule

$\langle E \rangle$ ::= $\langle T \rangle \langle TT \rangle$ TT.st := T.ptr; E.ptr := TT.ptr

$\langle TT_1 \rangle$ ::= + $\langle T \rangle \langle TT_2 \rangle$ $TT2$.st := mk_bin_op("+", TT_1.st, T.ptr); TT_1.ptr := TT_2.p

$\langle TT_1 \rangle$ ::= - $\langle T \rangle \langle TT_2 \rangle$ $TT2$.st := mk_bin_op("-", TT_1.st, T.ptr); TT_1.ptr := TT_2.pt

$\langle TT \rangle$ ::= ε TT.ptr := TT.st

$\langle T \rangle$ ::= $\langle F \rangle \langle FT \rangle$ FT.st := F.ptr; T.ptr := FT.ptr

$\langle FT_1 \rangle$ ::= * $\langle F \rangle \langle FT_2 \rangle$ FT_2.st := mk_bin_op("*", FT_1.st, F.ptr); FT_1.ptr := FT_2.pt

$\langle FT_1 \rangle$ ::= / $\langle F \rangle \langle FT_2 \rangle$ FT_2.st := mk_bin_op("/", FT_1.st, F.ptr); FT_1.ptr := FT_2.ptr

$\langle FT \rangle$ ::= ε FT.ptr := FT.st

$\langle F_1 \rangle$ ::= - $\langle F_2 \rangle$ F_1.ptr := mk_un_op("-", F_2.ptr)

$\langle F \rangle$ ::= ($\langle E \rangle$) F.ptr := E.ptr

$\langle F \rangle$ ::= unsigned_int F.ptr := mk_leaf(unsigned_int.val)

Example Decorated Parse Tree with AST

- Decorated parse tree of **(1+3)*2** with AST