Factoid

(with a grain of salt)

Language	Statements Ratio
C	1
C++	2.5
Fortran	2.5
Java	2.5
MS Visual Basic	4.5
Perl	6
Smalltalk	6
Python	6

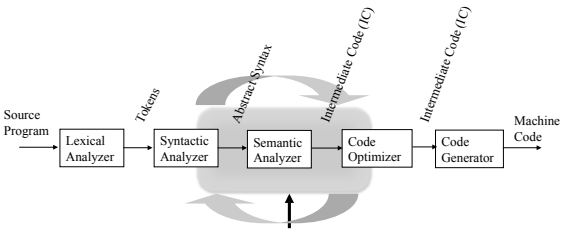
SR: ratios of source statements in several high-level languages to the equivalent code in C.

Semantics

What will happen when we do X?

Check rules and generate code

Fuzzy Boundaries and Interleavings



Questions on Semantics

- What happens when executing while(pred)?
- How is an expression evaluated?
- What has higher precedence ! or -?
- Is if(A&B) = if(B&A) ?
- What can I do if an object is serializable?
- Do variables have a default value?
- What can synchronize do to a schedule?

How hard can it be to specify such semantics?

http://www.**python**.org/doc/current/ref/binary.html

…

5.6 Binary arithmetic operations

The binary arithmetic operations have the conventional priority levels. Note that some of these operations also apply to certain non-numeric types. Apart from the power operator, there are only two levels, one for multiplicative operators and one for additive operators:

m_expr::=u_expr **|** m_expr **"*"** u_expr **|** m_expr **"//"** u_expr **|** m_expr **"/"** u_expr**|** m_expr **"%"** u_expr**a_expr::=**m_expr **|** a_expr **"+"** m_expr **|** a_expr **"-"** m_expr

The * (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, or one argument must be an integer (plain or long) and the other must be a sequence. In the former case, the numbers are converted to a common type and then multiplied together. In the latter case, sequence repetition is performed; a negative repetition factor yields an empty sequence.

The / (division) and // (floor division) operators yield the quotient of their arguments. The numeric arguments are first converted to a common type. …

Semantic Rules

- Static
  - Enforced by compiler
- Dynamic
  - Need is detected by compiler, code is added for enforcing it at runtime

Why do we need both types?

- Set by developers
  - Assertions (pre and post conditions, invariants)

Build Parse Tree

E → E + T
E → E - T
E → T
T → T * F
T → T / F
T → F
F → - F
F → (E)
F → const

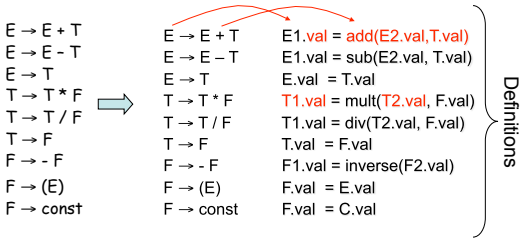
(1+3)*2

CFGs are not enough to do semantics checks

Try: $L = a^n b^n c^n$ = , *abc,* aabbcc, aaabbbccc, . . .

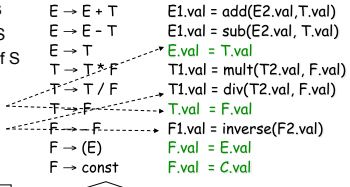
Attribute Grammars

- Associate meaning to program
- Provide framework to annotate trees



Attribute Grammars

- Attributes for symbols
  - *val* with each symbol S
  - S.val is the meaning of S
- Rules for productions
  - Copy rules
  - Semantic *functions*



Different kinds for different analysis:
- Denotational
- Axiomatic
- **Operational**

Rules and functions can be much more complex

Synthesized attributes: parameters on rhs of production, calculated on lhs

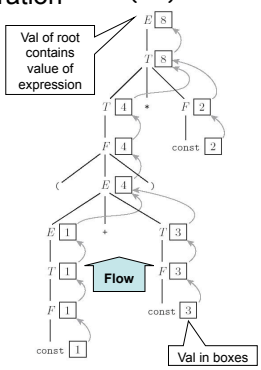
Evaluating Attributes

Annotation/Decoration

(1+3)*2

E → E + T E1.val = add(E2.val,T.val)
E → E - T E1.val = sub(E2.val, T.val)
E → T E.val = T.val
T → T * F T1.val = mult(T2.val, F.val)
T → T / F T1.val = div(T2.val, F.val)
T → F T.val = F.val
F → - F F1.val = inverse(F2.val)
F → (E) F.val = E.val
F → const F.val = C.val

Set by scanner

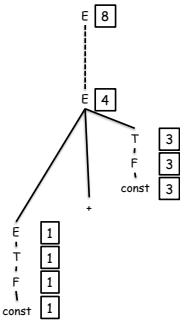


Evaluating Attributes

Annotation/Decoration

E → E + T E1.val = add(E2.val,T.val)
E → E - T E1.val = sub(E2.val, T.val)
E → T E.val = T.val
T → T * F T1.val = mult(T2.val, F.val)
T → T / F T1.val = div(T2.val, F.val)
T → F T.val = F.val
F → - F F1.val = inverse(F2.val)
F → (E) F.val = E.val
F → const F.val = C.val

…
reduce
reduce
reduce
3)*2
+3)*2
reduce
reduce
reduce
(1+3)*2



CFGs alone may be to complex

Modify so that it accepts programs with at least one “write” statement

1. *program* ⟶ *stmt_list* $$	program → other_stmt_list write expr stmt_list $$
2. *stmt_list* ⟶ *stmt_list stmt*	other_stmt_list → other_stmt_list other_stmt | ε
3. *stmt_list* ⟶ *stmt*	other_stmt → id := expr | read id
4. *stmt* ⟶ id := *expr*	stmt_list → stmt_list stmt | ε
5. *stmt* ⟶ read id	stmt → id := expr
6. *stmt* ⟶ write *expr*	stmt → read id
7. *expr* ⟶ *term*	stmt → write expr
8. *expr* ⟶ *expr add_op term*	expr → term
9. *term* ⟶ *factor*	expr → expr add_op term
10. *term* ⟶ *term mult_op factor*	term → factor
11. *factor* ⟶ (*expr*)	…
12. *factor* ⟶ id	
13. *factor* ⟶ number	
14. *add_op* ⟶ +	
15. *add_op* ⟶ -	
16. *mult_op* ⟶ *	
17. *mult_op* ⟶ /	

With attribute grammar

Modify so that it accepts programs with at least one “write” statement

1. *program* ⟶ *stmt_list* $$	program.ok = (write.val <= 1)
2. *stmt_list* ⟶ *stmt_list stmt*	
3. *stmt_list* ⟶ *stmt*	
4. *stmt* ⟶ id := *expr*	
5. *stmt* ⟶ read id	
6. *stmt* ⟶ write *expr*	write.val = write.val + 1
7. *expr* ⟶ *term*	
8. *expr* ⟶ *expr add_op term*	
9. *term* ⟶ *factor*	
10. *term* ⟶ *term mult_op factor*	
11. *factor* ⟶ (*expr*)	
12. *factor* ⟶ id	
13. *factor* ⟶ number	
14. *add_op* ⟶ +	
15. *add_op* ⟶ -	
16. *mult_op* ⟶ *	
17. *mult_op* ⟶ /	

CFG < Attributes Grammars

$L = a^n b^n c^n$ = , *abc,* aabbcc, aaabbbccc, . . . is not a CFG

Build S-attribute grammar
Associates a Boolean attribute ok with the root R of a parse tree if and only if string corresponding to fringe of tree is in *L*.

G → As Bs Cs	G.ok
As → a As	
As → ε	
Bs → b Bs	
Bs → ε	
Cs → c Cs	
Cs → ε	

CFG < Attributes Grammars

$L = a^n b^n c^n$ = , *abc,* aabbcc, aaabbbccc, . . . is not a CFG

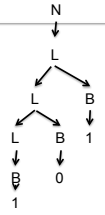
Build S-attribute grammar
Associates a Boolean attribute ok with the root R of a parse tree if and only if string corresponding to fringe of tree is in *L*.

G → As Bs Cs	G.ok = (As.val = Bs.val = Cs.val)
As → a As	As1.val = As2.val + 1
As → ε	As.val = 0
Bs → b Bs	Bs1.val = Bs2.val + 1
Bs → ε	Bs.val = 0
Cs → c Cs	Cs1.val = Cs2.val + 1
Cs → ε	Cs.val = 0

One more CFG with Attribute Grammar

B → 0	***B.val = 0***
B → 1	***B.val = 1***
L → B	***L.val = B.val; L.len = 1***
L → L B	***L1.val = 2 * L2.val + B.val; L1.len = L2.len + 1***
N → L	***N.val = L.val;*** **N.len = L.len**

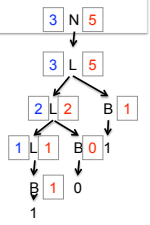
1. Write down the parse tree of: 101
2. Decorate parse tree



One more CFG with Attribute Grammar

B → 0	***B.val = 0***
B → 1	***B.val = 1***
L → B	***L.val = B.val; L.len = 1***
L → L B	***L1al = 2 * L2.val + B.val; L1.len = L2.len + 1***
N → L	***N.val = L.val; N.len = L.len***

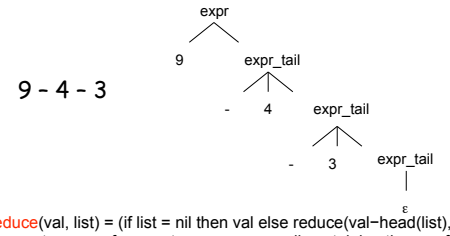
1. Write down the parse tree of: 101
2. Decorate parse tree



Limitation of S-attributes

Try to build S-attribute grammar
Challenge: left-associative

expr → const expr_tail
expr_tail → - const expr_tail
expr_tail → ε



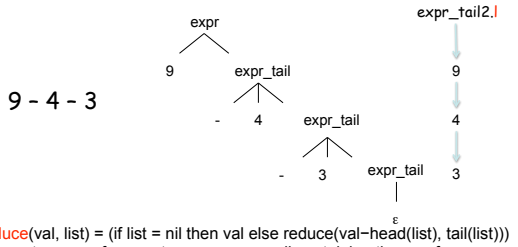
reduce(val, list) = (if list = nil then val else reduce(val−head(list), tail(list)))
cons returns a reference to a new cons cell containing those references

Limitation of S-attributes

Try to build S-attribute grammar
Challenge: left-associative

expr → const expr_tail
expr_tail → - const expr_tail
expr_tail → ε

expr.v = reduce(const.v, expr_tail.l)
expr_tail1.l = cons(const.v, expr_tail2.l)
expr_tail.l = nil



reduce(val, list) = (if list = nil then val else reduce(val−head(list), tail(list)))
cons returns a reference to a new cons cell containing those references

Types of attribute grammars

- **Synthesized attributes**
  - **Values based on attributes of descendents (child non-terminals in same production)**
- **Inherited attributes**
  - **Values based on attributes of parent (LHS non-terminal) or siblings (non-terminals on RHS of same production**

Alternative: Inherited Attributes

- Attributes define elements of RHS of grammar
- May depend on things above/side of in tree

expr → const expr_tail	expr_tail.st = const.val expr.val = expr_tail.val
expr_tail → - const expr_tail	expr_tail2.st = expt_tail1.st - const.val expr_tail1.val = expr_tail2.val
expr_tail → ε	expr_tail.val = expr_tail.st

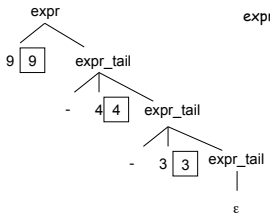
Alternative: Inherited Attributes

Get constant values from lexer

expr → const expr_tail
expr_tail.st = const.val
expr.val = expr_tail.val

expr_tail1 → - const expr_tail2
expr_tail2.st = expt_tail1.st - const.val
expr_tail1.val = expr_tail2.val

expr_tail → ε
expr_tail.val = expr_tail.st



9 - 4 - 3

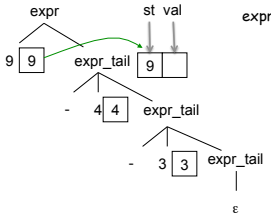
Alternative: Inherited Attributes

Copies value of expression so far into subtotal

expr → const expr_tail
expr_tail.st = const.val
expr.val = expr_tail.val

expr_tail1 → - const expr_tail2
expr_tail2.st = expt_tail1.st - const
expr_tail1.val = expr_tail2.val

expr_tail → ε
expr_tail.val = expr_tail.st



9 - 4 - 3

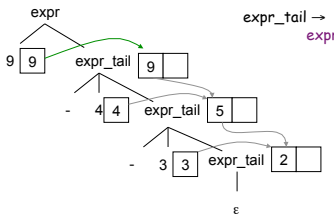
Alternative: Inherited Attributes

expr → const expr_tail
expr_tail.st = const.val
expr.val = expr_tail.val

Perform subtraction from the top down

expr_tail1 → - const expr_tail2
expr_tail2.st = expt_tail1.st – const
expr_tail1.val = expr_tail2.val

expr_tail → ε
expr_tail.val = expr_tail.st



9 – 4 – 3

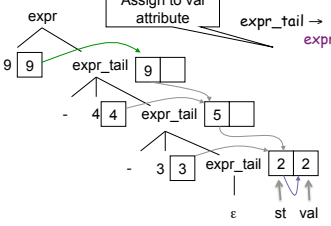
Alternative: Inherited Attributes

expr → const expr_tail
expr_tail.st = const.val
expr.val = expr_tail.val

expr_tail1 → - const expr_tail2
expr_tail2.st = expt_tail1.st – const.val
expr_tail1.val = expr_tail2.val

Assign to val attribute

expr_tail → ε
expr_tail.val = expr_tail.st



9 – 4 – 3

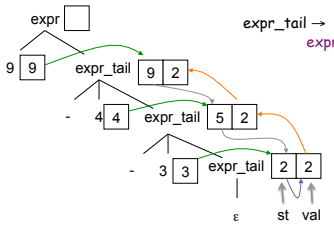
Alternative: Inherited Attributes

expr → const expr_tail
expr_tail.st = const.val
expr.val = expr_tail.val

Propagate upwards

expr_tail1 → - const expr_tail2
expr_tail2.st = expt_tail1.st – const.val
expr_tail1.val = expr_tail2.val

expr_tail → ε
expr_tail.val = expr_tail.st



9 – 4 – 3

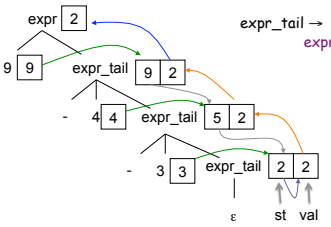
Alternative: Inherited Attributes

expr → const expr_tail
expr_tail.st = const.val
expr.val = expr_tail.val

Copies value rightmost leaf back up to root

expr_tail1 → - const expr_tail2
expr_tail2.st = expt_tail1.st – const.val
expr_tail1.val = expr_tail2.val

expr_tail → ε
expr_tail.val = expr_tail.st



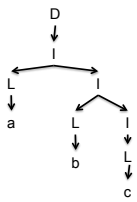
9 – 4 – 3

CFG and Attribute Grammar

Identifiers with no letters repeated (e.g., moon - illegal, money - legal)

D → I	***I.str= {}; D.val = I.val; accept if D.val != error***
I → L I	***L.str = I1.str; I2.str = L.val; I1.val = I2.val***
I → L	***L.str = I.str; I.val = L.val***
L → a | b | … | z	***L.val = concatenation of val returned by scanner to L.str if this is not a repeated letter, else error.***

Parse tree for abc

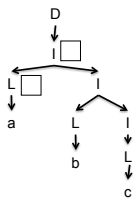


CFG and Attribute Grammar

Identifiers with no letters repeated (e.g., moon - illegal, money - legal)

D → I	***I.str= {}; D.val = I.val; accept if D.val != error***
I → L I	***L.str = I1.str; I2.str = L.val; I1.val = I2.val***
I → L	***L.str = I.str; I.val = L.val***
L → a | b | … | z	***L.val = concatenation of val returned by scanner to L.str if this is not a repeated letter, else error.***

Parse tree for abc

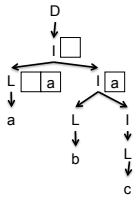


CFG and Attribute Grammar

Identifiers with no letters repeated (e.g., moon - illegal, money - legal)

D → I	*I.str= {}; D.val = I.val; accept if D.val != error*
I → L I	*L.str = I1.str; I2.str = L.val; I1.val = I2.val*
I → L	*L.str = I.str; I.val = L.val*
L → a | b | … | z	*L.val = concatenation of val returned by scanner to L.str if this is not a repeated letter, else error.*

Parse tree for abc

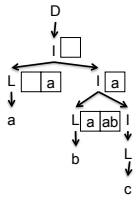


CFG and Attribute Grammar

Identifiers with no letters repeated (e.g., moon - illegal, money - legal)

D → I	*I.str= {}; D.val = I.val; accept if D.val != error*
I → L I	*L.str = I1.str; I2.str = L.val; I1.val = I2.val*
I → L	*L.str = I.str; I.val = L.val*
L → a | b | … | z	*L.val = concatenation of val returned by scanner to L.str if this is not a repeated letter, else error.*

Parse tree for abc

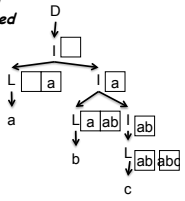


CFG and Attribute Grammar

Identifiers with no letters repeated (e.g., moon - illegal, money - legal)

D → I	*I.str= {}; D.val = I.val; accept if D.val != error*
I → L I	*L.str = I1.str; I2.str = L.val; I1.val = I2.val*
I → L	*L.str = I.str; I.val = L.val*
L → a | b | … | z	*L.val = concatenation of val returned by scanner to L.str if this is not a repeated letter, else error.*

Parse tree for abc

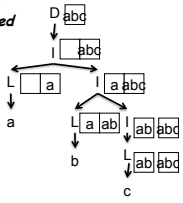


CFG and Attribute Grammar

Identifiers with no letters repeated (e.g., moon - illegal, money - legal)

D → I	*I.str= {}; D.val = I.val; accept if D.val != error*
I → L I	*L.str = I1.str; I2.str = L.val; I1.val = I2.val*
I → L	*L.str = I.str; I.val = L.val*
L → a | b | … | z	*L.val = concatenation of val returned by scanner to L.str if this is not a repeated letter, else error.*

Parse tree for abc



Attribute Flow

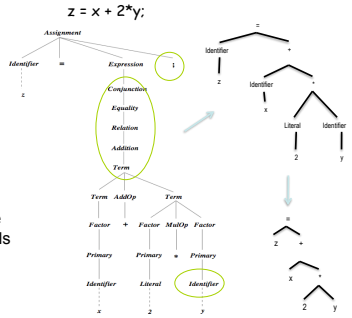
- Not defined by attribute grammar
  - Grammar is declarative – says how, no order
  - Any order will render same decoration
- If grammar is well-defined then
  1. Invoke semantic functions that have arguments defined
  2. Stop when no values changed

Action Routines

- A semantic function that tells compiler to execute at a particular parsing point
  - If semantic analysis is interleaved with parsing, then action routines perform semantic checks
  - Otherwise, action routines can be used to build a syntax tree

A common application of Action Routines: Transforming Parse Tree to Syntax Tree

- Redundant information
- Want to preserve meaning/shape
- Refinement
  - Remove separator/ punctuation terminal symbols
  - Remove all trivial (one child only) nonterminals
  - Replace remaining nonterminals with leaf terminals



A common application of Action Routines: Transforming Parse Tree to Syntax Tree

Pascal	C/C++

while i < n do begin
  i := i + 1;
end;
	while (i < n) {
	  i = i + 1;
	}

Both loops are designed to do the same thing
Machine code for both are the same
Minor differences in syntax

Essentials of looping construct in syntax tree are
1. Test expression to check when to stop looping
2. Body of the loop: statements to be repeated

Action Routines - Example

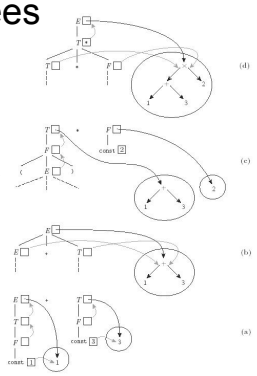
Points to nodes in syntax tree

$E \longrightarrow T$ { TT.st := T.ptr } TT { E.ptr := TT.ptr }
$TT_1 \longrightarrow + T$ { TT_2.st := make_bin_op ("+", TT_1.st, T.ptr) } TT_2 { TT_1.ptr := TT_2.ptr }
$TT_1 \longrightarrow - T$ { TT_2.st := make_bin_op ("−", TT_1.st, T.ptr) } TT_2 { TT_1.ptr := TT_2.ptr }
$TT \longrightarrow \epsilon$ { TT.ptr := TT.st }
$T \longrightarrow F$ { FT.st := F.ptr } FT { T.ptr := FT.ptr }
$FT_1 \longrightarrow * F$ { FT_2.st := make_bin_op ("×", FT_1.st, F.ptr) } FT_2 { FT_1.ptr := FT_2.ptr }
$FT_1 \longrightarrow / F$ { FT_2.st := make_bin_op ("÷", FT_1.st, F.ptr) } FT_2 { FT_1.ptr := FT_2.ptr }
$FT \longrightarrow \epsilon$ { FT.ptr := FT.st }
$F_1 \longrightarrow - F_2$ { F_1.ptr := make_un_op ("+/−", F_2.ptr) }
$F \longrightarrow (E)$ { F.ptr := E.ptr }
$F \longrightarrow$ const { F.ptr := make_leaf (const.ptr) }

Evaluating Attributes Syntax Trees

$E_1 \longrightarrow E_2 + T$
  ▷ E_1.ptr := make_bin_op("+", E_2.ptr, T.ptr)
$E_1 \longrightarrow E_2 - T$
  ▷ E_1.ptr := make_bin_op("−", E_2.ptr, T.ptr)
$E \longrightarrow T$
  ▷ E.ptr := T.ptr
$T_1 \longrightarrow T_2 * F$
  ▷ T_1.ptr := make_bin_op("×", T_2.ptr, F.ptr)
$T_1 \longrightarrow T_2 / F$
  ▷ T_1.ptr := make_bin_op("÷", T_2.ptr, F.ptr)
$T \longrightarrow F$
  ▷ T.ptr := F.ptr
$F_1 \longrightarrow - F_2$
  ▷ F_1.ptr := make_un_op("+/−", F_2.ptr)
$F \longrightarrow (E)$
  ▷ F.ptr := E.ptr
$F \longrightarrow$ const
  ▷ F.ptr := make_leaf(const.val)

(1+3)*2



TODO

- Complete read of Ch4
  - Except 4.5 and 4.6
  - Exercises 4.1 – 4.4, 4.7, 4.9
- Skip Ch5
- Continue with assignment #2 (Lex & Yacc)