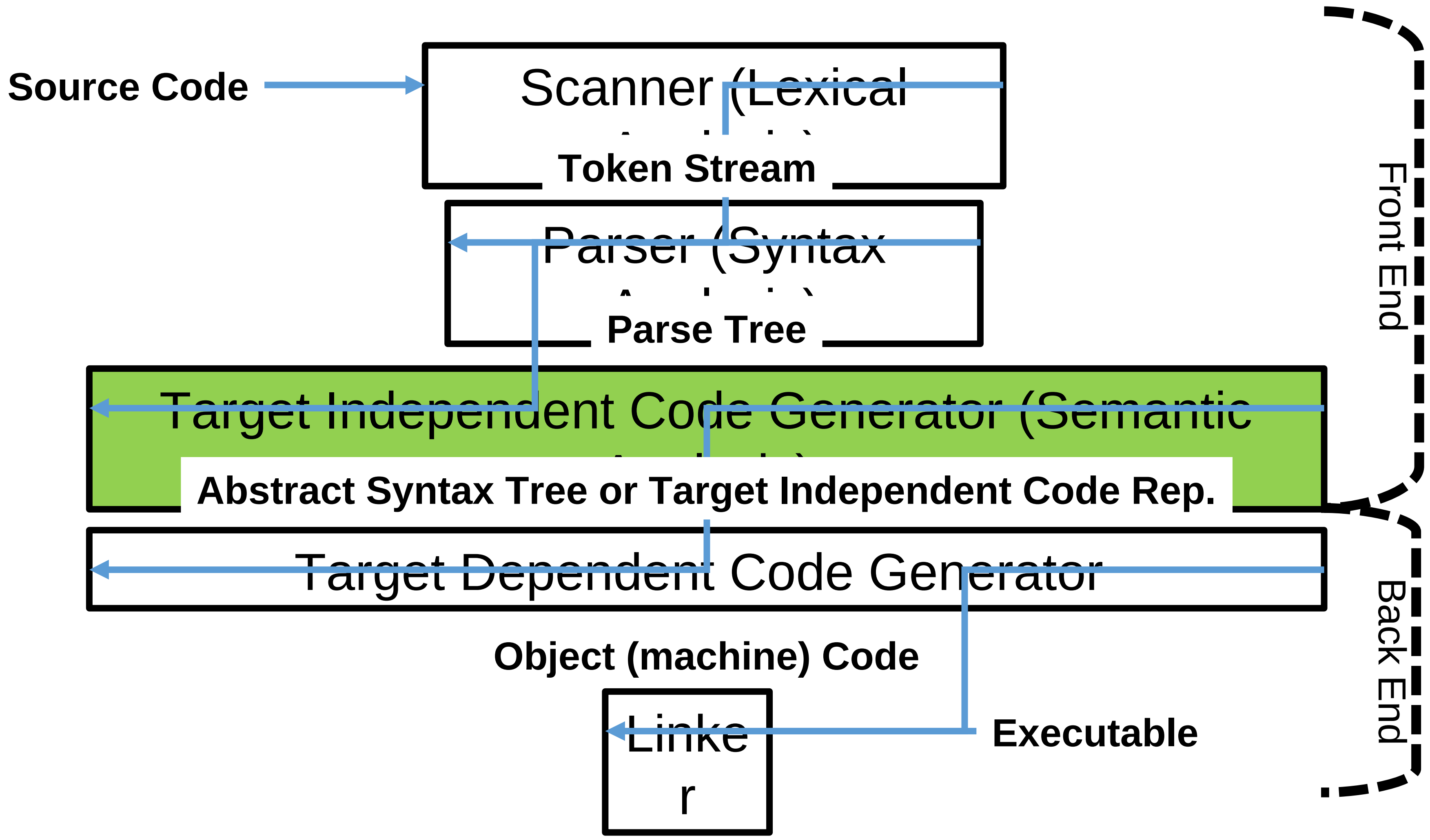
Semantic Analysis and Attribute Grammars

CSCI 3136: Principles of Programming Languages

Agenda

- S-Attributed and L-Attributed Grammars
- Examples
- Action Routines

Recall: Phases of Compilation


Source Code
Scanner (Lexical Analysis)
Token Stream
Parser (Syntax Analysis)
Parse Tree
Target Independent Code Generator (Semantic Analysis)
Abstract Syntax Tree or Target Independent Code Rep.
Target Dependent Code Generator
Object (machine) Code
Linker
Executable
Front End
Back End

Example 1: L = {anbncn|n ≥ 0}

- This is not a context free language, but can be specified by an attribute grammar

CFG w/ Labeled Symbols	Semantic Rules
S → A1 B1 C1	□ **if** A1.count != B1.count **or** A1.count != C1.count, **error**
A → A1 a	□ A.count = A1.count + 1
A → ε	□ A.count = 0
B → B1 b	□ B.count = B1.count + 1
B → ε	□ B.count = 0
C → C1 c	□ C.count = C1.count + 1
C → ε	□ C.count = 0

Symbol	Attributes
A	count : int
B	count : int
C	count : int

- Example: Consider parsing: aaaabbbbcccc

Example 2: L = {□^{H} {a,b,c}*: |□|a=|□|b=|□|c}

CFG w/ Labeled Symbols	Semantic Rules
S → X1	□ **if** X1.aCount != X1.bCount **or** X1.aCount != X1.cCount, **error**
X → a X1	□ X.aCount = X1.aCount + 1; X.bCount = X1.bCount; X.cCount = X1.cCount;
X → b X1	□ X.bCount = X1.bCount + 1; X.aCount = X1.aCount; X.cCount = X1.cCount;
X → c X1	□ X.cCount = X1.cCount + 1; X.bCount = X1.bCount; X.aCount = X1.aCount;
X → ε	□ X.aCount = 0; X.bCount = 0; X.cCount = 0;

Symbol	Attributes
X	aCount : int bCount : int cCount : int

Why do we need the **S** → **X** production?

Types of Attributes

- The previous examples are of *synthesized* (bottom up) attribute grammars.
- There are two types of Attributes
    - ***Synthesized* attributes** are computed in the RHS and stored in LHS
    - ***Inherited* attributes** are computed using LHS and RHS and used by symbols further to the right.

Example 3: L = {anbncn|n ≥ 0}

- Using inherited attributes instead of synthesized.

CFG w/ Labeled Symbols	Semantic Rules
S → A1 B1 C1	☐ B1.iCount = A1.count; C1.iCount = A.count
A → A1 a	☐ A.count = A1.count + 1
A → ε	☐ A.count = 0
B → B1 b	☐ B1.iCount = B.iCount - 1
B → ε	☐ **if** B.iCount != 0, **error**
C → C1 c	☐ C1.iCount = C.iCount -
C → ε	☐ **if** C.iCount != 0, **error**

Symbol	Attributes
A	count : int
B	**iCount : int**
C	**iCount : int**

- Example: Consider parsing: aaaabbbbcccc

Example 4: Using Inherited Attributes

$L = \{□ \in \{a, b, c\}^*: |□|_a = |□|_b = |□|_c\}$

CFG w/ Labeled Symbols	Semantic Rules
S → X1	□ X1.aCount = 0; X1.bCount = 0; X1.cCount = 0;
X → a X1	□ X1.aCount = X.aCount + 1; X1.bCount = X.bCount; X1.cCount = X.cCount;
X → b X1	□ X1.bCount = X.bCount + 1; X1.aCount = X.aCount; X1.cCount = X.cCount;
X → c X1	□ X1.cCount = X.cCount + 1; X1.bCount = X.bCount; X1.aCount = X.aCount;
X → ε	□ **if** X.aCount != X.bCount **or** X.aCount != X.cCount , **error**

Symbol	Attributes
X	aCount : int bCount : int cCount : int

Recap

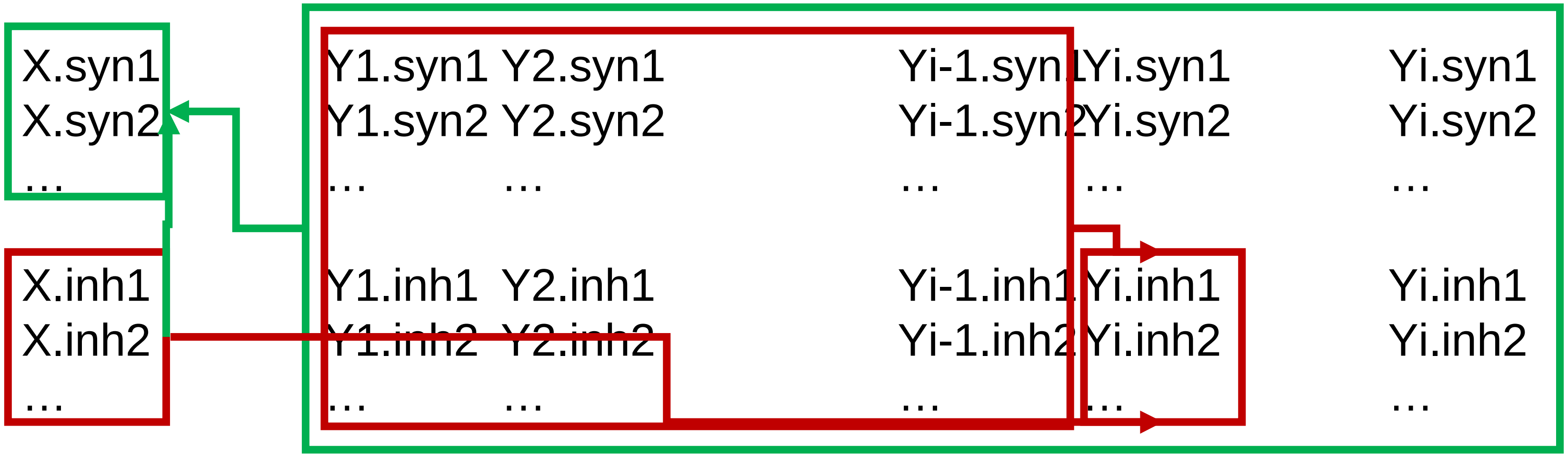
- Parse trees can be annotated or decorated with attributes and rules, which are executed as the tree is traversed.
- Synthesized attributes
  - Attributes of LHS of production are computed from attributes of RHS
  - Attributes flow bottom-up in the parse tree.
- Inherited attributes
  - Attributes in RHS are computed from attributes of LHS and symbols in RHS preceding them.
  - Attributes flow top-down in the parse tree.

S-Attributed and L-Attributed Grammars

- S-attributed grammar
  - All attributes are synthesized.
  - Attributes flow bottom-up.
- L-attributed grammar
  - Variables have both inherited and synthetic attributes
  - For each production $X \rightarrow Y1Y2 \ldots Yk$,
    - X.syn depends on
      - X.inh
      - Y1.inh, Y1.syn, Y2.inh, Y2.syn, . . . Yk.inh, Yk.syn
  - For all $1 \le i \le k$, Yi.inh depends on

Data Flow in L-Attributed Grammars

X → Y1Y2 . . . Yi-1Yi ...Yk



Computing L-Attributed Grammars

```
execute_rules( Node t, Node [] left_sibs ):
  # Don’t use t.synthetic and t.parent.synthetic
  t.compute_inherited( t.parent, left_sibs )


  children = []
  for each child of t:
    execute_rules( child, children )
```

Motivation: Why are they useful?

- In many cases context free grammars that capture associativity rules are not LL(1)
- We can rewrite the grammars to be LL(1) but…
- Resulting grammars do no capture associativity rules
- So, use attribute (L-attributed) grammars to capture the associativity rules.

Example: Left Associative Grammar

- Grammar
  - E → E A T
  - E → T
  - T → Int
  - A → +
  - A → −
- Parsing the expression

  5 − 2 + 3 illlustrates left associativity: (5 − 2) + 3
- This grammar is not

Predictor Table	
Production	**Predictor Set**
E → EAT	{Int}
E → T	{Int}
T → Int	{Int}
A → +	{+}
A → -	{-}

Example: Refactored Grammar

- Grammar
  - E → T E’
  - E’ → ε
  - E’ → A T E’
  - T → Int
  - A → +
  - A → −
- Parsing the expression

  5 − 2 + 3 illlustrates

Predictor Table	
Production	**Predictor Set**
E → T E’	{Int}
E’ → ε	{ε}
E’ → A T E’	(+, -}
T → Int	{Int}
A → +	{+}
A → -	{-}

Use an L-Attributed Grammar to Fix Left Associati

Sym	Attributes
E	val : int
E’	val : int **op : int**
T	val : int
A	func : operation
Int	val : String

Idea: Carry forward the left most computed value to ensure left associativity.

- Try parsing: 5 - 2 + 3

Labeled CFG	Semantic Rules
E → T E’1	□ E1′.op = T1.val; E.val = E′.val
E’ → ε	□ E′.val = E′.op
E’ → A1 T1 E’ 1	□ E1′.op = A.func(E′.op, T1.val); E′.val = E1′.val
T → Int1	□ T.val = Str2Int(Int1.val)
A → +	□ A.func = add
A → -	□ A.func = add

Example: Error Checking

Labeled CFG	Semantic Rules
Assignment → LValue1 ‘=‘ Expr1	□ **if not** assignable(Lvalue1.t, Expr1.t), **error**
LValue → Id1 ArrIdx1	□ **if not** declared(Id1.name), **error** □ **if not** indexable(Id1.name, ArrIdx1.dim), **error**
ArrIdx → ε	□ ArrIdx.dim = 0
ArrIdx → ‘[‘ Expr1 ‘]’ ArrIdx1	□ **if not** isType(Expr1.t, Integer), **error** □ ArrIdx.dim = ArrIdx1.dim + 1

Sym	Attributes
Assignment	
LValue	t : Type
Id	name : String
ArrIdx	dim : int
Expr	t : Type

Example: Generate Java Code

Labeled CFG	Semantic Rules
E → E1A1T1	□ E.tmp = tmpSeqNum++ **output(** “int tmp%d = tmp%d %s %s;”, E.tmp, E1.tmp, A1.op, T1.var **)**
E → T1	□ E.tmp = tmpSeqNum++ **output(** “int tmp%d = %s;”, E.tmp, T1.var **)**
T→ Id1	□ T.var = id1.name
A → ‘+’	□ A.op = “+”
A→ ‘-’	□ A.op = “-”

Sym	Attributes
E	tmp : int
T	var : String
Id	name : String
A	op : String

Try generating Java code for the expression: a + b - c

Action Routines

- Action routines are instructions for ad-hoc translation interleaved with parsing
- Parser generators allow programmers to specify action routines as part of the grammar
- Action routines can appear anywhere in a rule (as long as the grammar is LL(1)).
- Example
    - E1 → A T **{E2.op = A.fun(E1.op,T.val)}** E2 **{E1.val = E2.val}**
- Action routines are supported, for example, in yacc and bison