CS 5363, Fall 2013
Attribute Grammars

1 Reading / Source Material

- Cooper: Chapter 4
- Scott: Chapter 4

2 Objectives: Be able to . . .

1. Given a simple attributed grammar and a sentence in the language described by the grammar, calculate the attributes.
2. Given an attributed grammar, identify attributes of non-terminals for which the attribute is always inherited
3. Given an attributed grammar, identify attributes of non-terminals for which the attribute is always synthesized
4. Given an expression grammar in which the productions for left-associative operations have undergone recursion elimination to produce an LL(1) grammar, be able to write an L-attribute grammar that evaluates the expression applying left-associativity.
5. write an L-attributed grammar that generates an AST for a typical procedural programming language described by a BNF and informal semantics
6. given an L-attributed grammar, write a recursive descent parser with ad-hoc rules implementing the attributed grammar
7. write a tree grammar to type-check the AST of programs written in a typical procedural programming language described by a BNF, informal semantics, and informal type rules

3 Outline

1. Semantic Elaboration ("Context-Sensitive" Analysis)
   (a) beyond syntax, beyond context-free grammars
   (b) examples:
      i. declaration before use
      ii. number of parameters in procedural call
      iii. types of variables / expressions
      iv. return statements
   (c) purpose:
      i. discover information needed to generate efficient code

        ii. detect errors

2. Attribute Grammar

    (a) context free grammar + attributes + semantic functions
    (b) expression example
    (c) synthesized attributes
        i. rules defining the attribute, define it for the non-terminal on the left-hand side (lhs) of the production in which the rule occurs
        ii. that is, the attribute of a non-terminal is defined in terms of the non-terminal's own attributes and those of its children
    (d) inherited attributes
        i. rules defining the attribute, define it for a non-terminal on the right-hand side (rhs) of the production in which the rule occurs
        ii. that is, the attribute of a non-terminal is defined in terms of the non-terminal's own attributes, those of its parents, and those of its siblings
    (e) S-attributed grammars
        i. an attribute grammar where all attributes are synthesized
    (f) L-attributed grammars
        i. an attribute grammar where all attributes are inherited or synthesized and satisfy the following additional constraints:
            A. the synthesized attributes of a lhs non-terminal depend only on inherited attributes of that lhs non-terminal and on (synthesized or inherited) attributes of the rhs non-terminals.
            B. the inherited attributes of a rhs non-terminal depend only on inherited attributes of the lhs non-terminal and on (synthesized or inherited) attributes of rhs non-terminals that occur to the left of the non-terminal for which the attribute is being defined.
        ii. relation to recursive descent parser
            A. inherited attributes can be passed as parameters
            B. synthesized attributes can be returned as values

3. tree grammars

4 Vocabulary

Attribute Grammar, attribute, synthesized attribute, inherited attribute, L-attributed grammar, S-attributed grammar

5 Questions

1. Consider the following grammar describe the binary numbers that are multiples of four.

```
<Number> -> 0 | <HighestBit>  <List>  <DoubleZero>
<HighestBit> -> 1
<DoubleZ> -> 00
<List> -> 1 <List> | 0 <List> |
```

For example, 1000 is a binary number satisfying the above grammar. Now, suppose we change it into an attribute grammar with the following attributes, such that 1000's value is 8 and its HighestBit's position is 3, that is, the position of the rightmost 0 is 0.

```
Symbol      Attribute
--------------------------
<Number>      value
<List>        value, position
<HighestBit>  position
```

(a) Fill in the Attribution Rules for each production.

```
Production                                      Attribution Rules
--------------------------------------------------------------------------------
1 <Number> -> 0                                 Number.value :=
--------------------------------------------------------------------------------
2 <Number> -> <HighestBit> <List> <DoubleZ>     HighestBit.position :=
                                                Number.value :=
--------------------------------------------------------------------------------
3 <List0>  -> 1 <List1>                         List0.position :=
                                                List0.value :=
--------------------------------------------------------------------------------
4 <List0>  -> 0 <List1>                         List0.position :=
                                                List0.value :=
--------------------------------------------------------------------------------
5 <List>   ->                                  List.value :=
                                                List.position :=
--------------------------------------------------------------------------------
```

(b) Using the above attribute grammar, build the syntax tree for the binary number 101100, annotating all attributes with the corresponding value.

(c) Indicate which attributes are inherited and which are synthesized.

(d) Is your attributed grammar L-attributed?

2. Given the attribute grammar:

```
<A> ::= <B>
^ A.x = B.x
^ B.y = 0

<B1> ::= c <B2>
```

```
^ B1.x = B2.x + 1
^ B2.y = B1.y + 1

<B> ::=
^ B.x = 0
```

(a) Which attributes are synthesized and which are inherited?
(b) Draw a attribute-annotated parse tree for the sentence: ccc
Add arrows showing how attribute information flows between pairs of attributes.

3. Consider the attribute grammar:

```
<A> ::= x <B>
^ A.result = B.result
<B1> ::= y <B2>
^ B1.result = B2.result + 2
<B1> ::= z <B2>
^ B1.result = B2.result + 1
<B> ::=
```

(a) Which attributes are synthesized and which are inherited?
(b) Draw an attribute-annotated parse tree for teh sentence: xzyy
Add arrows showing how attribute information flows between pairs of attributes.

4. Consider the following attributed tree grammar for evaluating an expression with an integer value of X whose value is 5 and Y whose value is 3.

```
start: <start> ::= <expr>
^ <expr>.env = <'X', 5, <'Y', 3, nil>>

id: <expr> ::=
^ <expr>.val = lookup(id.$literalText$)

int_const: [expr] ::=
^ <expr>.val = text2Integer(int_const.$literalText$)

'+': <expr1> ::= <expr2> <expr3>
^ <expr2>.symtab = <expr1>.symtab
^ <expr3>.symtab = <expr1>.symtab
^ <expr1>.val = <expr2>.val + <expr3>.val

'*': <expr1> ::= <expr2> <expr3>
^ <expr2>.symtab = <expr1>.symtab
^ <expr3>.symtab = <expr1>.symtab
^ <expr1>.val = <expr2>.val * <expr3>.val
```

Assume that int_const and id nodes have a pre-defined attribute $literalText$ that corresponds to the text of the id or integer constant, respectively.

Also assume an auxiliary function lookup, defined as follows:

```
  lookup(X, env) = if env = nil
                     then error
                     else let <Y, n env2> = env in
                            if Y = X then n else lookup(X, env2)
                   end
```