CSE443
Compilers

Dr. Carl Alphonse
alphonce@buffalo.edu
343 Davis Hall
Phases of a compiler

Semantic analysis
How do rules for function definitions work?

\[
\text{type identifier `:` pblock `->` identifier} \\
\text{function identifier}_1 `:` identifier}_2 \text{sblock}
\]

*Function type.*

*Function definition.*

*identifier}_1 \text{ is function name,}*

*identifier}_2 \text{ is function type*}
How do functions work?

A function maps a value from its domain to a value in its range:

\[ \text{domain} \quad \xrightarrow{f} \quad \text{range} \]

\[ x \quad f(x) \]
Project Notes

In code:

```cpp
double f(int x) { ... }
```
Back to our language:

**type** identifier `:` pblock `'->'` identifier  
*Function type.*

**function** identifier₁ `:` identifier₂ sblock  
*Function definition.*  
 identifiers₁ is function name,  
 identifiers₂ is function type
Project Notes

In code:

\[
\text{type foo : (int x) -> double}
\]

\[
\text{function f : foo [ ... ]}
\]
sblock is:
    '{' ( dblock ) statement-list '}

dblock is:
    '[' declaration-list ']

declaration-list is:
    declaration ';' declaration-list | declaration

declaration is:
    identifier ':' identifier-list  \textit{LHS is type, RHS is list of variable names.}

identifier-list is:
    identifier ( assignOp constant ) ',' identifier-list | identifier ( assignOp constant )
How do declarations work?

{ 
  [ integer : a := 3, b ;
    real : x, y := 0.2, z := -5.1 ]

  ... statements ...
}

What is permitted as an initializer?
Project Notes

How do declarations work?

{  
[ integer : a := 3, b ;  
   real : x, y := 0.2, z := -5.1 ]
...

... statements ...

}  

Why only constants as initializers?

How do nested scopes work in e.g. Java?
Attribute grammars

Attribute grammars provide a neater way of encoding such information.

Each syntactic rule of the grammar can be decorated with:

– a set of semantic rules/functions
– a set of semantic predicates
Attributes

- We can associate with each symbol X of the grammar a set of attributes A(X). Attributes are partitioned into:

  synthesized attributes S(X) – pass info up parse tree

  inherited attributes I(X) – pass info down parse tree
Example

<assign> → <var> = <expr>
<expr>.expType ← <var>.actType
<expr>.[2] + <var>.[3]
<expr>.actType ← if (var.[2].actType = int) and
               (var.[3].actType = int)
               then int
               else real
<expr>.actType == <expr>.expType

<expr> → <var>
<expr>.actType ← <var>.actType
<expr>.actType == <expr>.expType

<var> → A | B | C
<var>.actType ← lookUp(<var>.string)

Review

Syntactic rule
Semantic rule/function
Semantic predicate
Suppose:

- $A$ is int
- $B$ is int

The effects of the syntactic rules are shown in red.

Everything works!
This is the same example structure, but now assume A is of type real and B is of type int.

Actual type = real
Actual type = real
Actual type = int

Expected type = real
Actual type = real
Actual type = real

Need for type coercion recognized during ‘+’: int → real

Generate code to do conversion.

Suppose:
A is real
B is int
Suppose:
A is int
B is real

Houston, we have a problem!
Semantic predicate is \textbf{false}.

Generate error message.
Syntax-Directed Definitions

"A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions" [p. 304]
Syntax-Directed Definitions

"A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions" [p. 304]

See the 'union' type in the bison docs.
A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions." [p. 304]
Evaluation Orders for SDD's

"The dependency graph characterizes the possible orders in which we can evaluate the attributes at the various nodes of the parse tree. If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N." [p. 312]
Example: declaration grammar (Figure 5.4)

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
</table>
| 1 \( T \rightarrow F T' \) | \( T'.inh = F.val \)  
  \( T.val = T'.syn \) |
| 2 \( T' \rightarrow * F T_{1}' \) | \( T_{1}' .inh = T'.inh * F.val \)  
  \( T'.syn = T_{1}' .syn \) |
| 3 \( T' \rightarrow \epsilon \) | \( T'.syn = T'.inh \) |
| 4 \( F \rightarrow \text{digit} \) | \( F.val = \text{digit . lexval} \) |
**Example: declaration grammar**
(Figure 5.4)

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
</table>
| 1) $T \rightarrow FT'$ | $T'.inh = F.val$  
$T.val = T'.syn$ |
| 2) $T' \rightarrow *FT_1'$ | $T_1'.inh = T'.inh \times F.val$  
$T'.syn = T_1'.syn$ |
| 3) $T' \rightarrow \epsilon$ | $T'.syn = T'.inh$ |
| 4) $F \rightarrow digit$ | $F.val = digit.lexval$ |

Figure 5.4: An SDD based on a grammar suitable for top-down parsing
Example:
Annotated parse tree
(Figure 5.5)

Figure 5.5: Annotated parse tree for $3 \times 5$
<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
</table>
| 1) $T \rightarrow FT'$ | $T'.inh = F.val$
         | $T.val = T'.syn$ |
| 2) $T' \rightarrow *FT_1'$ | $T'.inh = T'.inh \times F.val$
         | $T'.syn = T'.syn$ |
| 3) $T' \rightarrow \epsilon$ | $T'.syn = T'.inh$ |
| 4) $F \rightarrow \text{digit}$ | $F.val = \text{digit.lexval}$ |

Figure 5.4: An SDD based on a grammar suitable for top-down parsing.

Figure 5.5: Annotated parse tree for $3 \times 5$. 

Using attributes of node 3.
Example: Dependency graph (Figure 5.7)

Figure 5.7: Dependency graph for the annotated parse tree of Fig. 5.5
Synthesized and Inherited attributes

"A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself." [p. 304]

"An inherited attribute at node N is defined only in terms of attribute values at N's parents, N itself, and N's siblings." [p. 304]
S-Attributed Definitions

"The first class [of SDD's that do not permit dependency graphs with cycles] is defined as follows:

An SDD is S-attributed if every attribute is synthesized." [p. 313]
L-Attributed Definitions

"The second class of SDD's [that do not permit dependency graphs with cycles] is called L-attributed definitions. The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left (hence 'L-attributed')." [p. 313]
Semantic rules/functions

- We can associate with each rule R of the grammar a set of semantic functions.

- For rule $x_0 \rightarrow x_1 x_2 \ldots x_n$
  - synthesized attribute of LHS:
    $$S(x_0) = f(A(x_1), A(x_2), \ldots, A(x_n))$$
  - inherited attribute of RHS member:
    for $1 \leq j \leq n$, $I(x_j) = f(A(x_0), \ldots, A(x_{j-1}))$
    (note that dependence is on siblings to left only)
L-Attributed Definitions

"Each attribute must be either:

1. Synthesized, or

2. Inherited, but with the rules limited as follows. Suppose that there is a production $A \rightarrow X_1 X_2 \ldots X_n$ and that there is an inherited attribute $X_i.a$ computed by a rule associated with this production. Then the rule may use only:

(a) Inherited attributes associated with the head $A$.

(b) Either inherited or synthesized attributes associated with the occurrences of symbols $X_1, X_2, \ldots X_{i-1}$ located to the left of $X_i$.

(c) Inherited or synthesized attributes associated with this occurrence of $X_i$ itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this $X_i$. ” [p. 313-4]
L-Attributed Definitions

"Each attribute must be either:

1. Synthesized, or

2. Inherited, but with the rules limited as follows. Suppose that there is a production $A \rightarrow X_1 X_2 \ldots X_n$ and that there is an inherited attribute $X_i.a$ computed by a rule associated with this production. Then the rule may use only:

   (a) Inherited attributes associated with the head $A$.

   (b) Either inherited or synthesized attributes associated with the occurrences of symbols $X_1, X_2, \ldots X_{i-1}$ located to the left of $X_i$.

   (c) Inherited or synthesized attributes associated with this occurrence of $X_i$ itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this $X_i$. " [p. 313-4]
Reading assignment

Sections 5.3 and 5.4 (pages 318 through 337).
Example 5.19 (p. 335)

(we'll revisit in 6.6.3 on page 401)

$ \rightarrow \text{while ( C ) } S_1$

What are the semantics of this?
Example 5.19 (p. 335)

(we’ll revisit in 6.6.3 on page 401)

$ S \rightarrow \text{while ( } C \text{ ) } S_1 $

What are the semantics of this?

$L_1 = \text{new}()$
$L_2 = \text{new}()$
$S_1.next = L_1$
$C.false = S.next$
$C.true = L_2$
$S.code = \text{label} || L_1 || C.code || \text{label} || L_2 || S_1.code$
Example 5.19 (p. 335)

(we'll revisit in 6.6.3 on page 401)

\[ S \rightarrow \text{while ( C ) } S_1 \]

There are multiple jumps out of C on both true and false to show potential short-circuiting.