CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall
Homework 2 and Project 2
due date adjustment

Wednesday March 11 @ 5:00 PM

Homework 3 and Project 3

Homework 3 released Wednesday
Project 3 released Wednesday
Homework 2 and Project 2 due date adjustment?

What would be helpful?
Phases of a compiler

Semantic analysis

Figure 1.6, page 5 of text
Overview

§5.3 Applications of Syntax-Directed Translation

- type checking
- code generation
"Syntax-directed translation schemes are a complementary notation to syntax-directed definitions. [...] A syntax-directed translation scheme (SDT) is a context-free grammar with program fragments embedded within production bodies." [p. 324]
Syntax-Directed Translation Schemes

"Any SDT can be implemented by first building a parse tree and then performing the actions in a [...] pre-order traversal." [p. 324]

"Typically, SDT's are implemented during parsing, without building a parse tree." [p. 324]
Syntax-Directed Translation Schemes

"...the simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDD is S-attributed. In that case, we can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production." [p. 324]
"If the attributes are all synthesized, and the actions occur at the ends of the productions, then we can compute the attributes for the head when we reduce the body to the head." [p. 325]
"We consider [now] the more general case of an L-attributed SDD." [p. 331]

"The rules for turning an L-attributed SDD into an SDT are as follows:

1. Embed the action that computes the inherited attributes for a nonterminal \( A \) immediately before the occurrence of \( A \) in the body of the production.

2. Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production." [p. 331]
Implementing L-Attributed SDD's

"...we discuss the following methods for translating during parsing:

6. Implement an SDT in conjunction an LR parser. ... since the SDT for an L-attributed SDD typically has actions in the middle of productions, and we cannot be sure during an LR parse that we are even in that production until its entire body has been constructed ... [however] if the underlying grammar is LL, we can always handle both the parsing and translation bottom-up." [p. 338]
Bottom-up parsing of L-Attributed SDD’s

"...given an L-attributed SDD on an LL grammar, we can adapt the grammar to compute the same SDD on the new grammar during an LR parse” [p. 348]

1. "Start with the SDT […] which places embedded actions before each nonterminal to compute its inherited attributes and an action at the end of the production to compute synthesized attributes.

2. Introduce into the grammar a marker nonterminal in place of each embedded action. Each such place gets a distinct marker, and there is one production for any marker M, M -> \( \varepsilon \).

3. Modify the action \( a \) if marker nonterminal M replaces it in some production A -> \( \alpha \{a\} \beta \), and associate with M -> \( \varepsilon \) an action \( a' \) that

   (a) Copies, as inherited attributes of M, any attributes of A or symbols of \( \alpha \) that action \( a \) needs.

   (b) Computes the attributes in the same way as \( a \), but makes those attributes be synthesized attributes of M” [p. 349]
Bottom-up parsing of L-Attributed SDD's

"...given an L-attributed SDD on an LL grammar, we can adapt the grammar to compute the same SDD on the new grammar during an LR parse" [p. 348]

1. "Start with the SDT [...] which places embedded actions before each nonterminal to compute its inherited attributes and an action at the end of the production to compute synthesized attributes.

2. Introduce into the grammar a marker nonterminal in place of each embedded action. Each such place gets a distinct marker, and there is one production for any marker M, M → ε.

3. Modify the action a if marker nonterminal M replaces it in some production A → α{a}β, and associate with M → ε an action a' that

(a) Copies, as inherited attributes of M, any attributes of A or symbols of α that action a needs.

(b) Computes the attributes in the same way as a, but makes those attributes be synthesized attributes of M" [p. 349]

"This change appears illegal, since typically the action associated with production M → ε will have access to grammar symbols that do not appear in this production." [p. 349]
Bottom-up parsing of L-Attributed SDD's

"...given an L-attributed SDD on an LL grammar, we can adapt the grammar to compute the same SDD on the new grammar during an LR parse" [p. 348]

1. "Start with the SDT [...] which places embedded actions before each nonterminal to compute its inherited attributes and an action at the end of the production to compute synthesized attributes.

2. Introduce into the grammar a marker nonterminal in place of each embedded action. Each such place gets a distinct marker, and there is one production for any marker $M$, $M \rightarrow \varepsilon$.

3. Modify the action $a$ if marker nonterminal $M$ replaces it in some production $A \rightarrow \alpha \{a\} \beta$, and associate with $M \rightarrow \varepsilon$ an action $a'$ that

(a) Copies, as inherited attributes of $M$, any attributes of $A$ or symbols of $\alpha$ that action $a$ needs.

(b) Computes the attributes in the same way as $a$, but makes those attributes be synthesized attributes of $M$" [p. 349]

"...we shall implement the actions on the LR parsing stack, so the necessary attributes will always be available a known number of positions down the stack." [p. 349]
Consider production $A \rightarrow B \ C$, with inherited attribute of $B$ computed as a function of an inherited attribute of $A$:

$$A \rightarrow \{ \ B.i = f(A.i); \ \} \ B \ C$$

To parse bottom up, introduce marker non-terminal:

$$A \rightarrow \ M \ B \ C$$

$$M \rightarrow \{ \ M.i = A.i; \ M.s = f(M.i); \ \}$$
Example 5.25 [p. 349]

A → M B C
M → \{ M.i = A.i; M.s = f(M.i); \}

"Notice that the rule for M does not have A.i available to it, but in fact we shall arrange that every inherited attribute for a non-terminal such as A appears on the stack immediately below where the reduction to A will later take place. Thus, when we reduce ε to M, we shall find A.i immediately below it, from where it may be read. Also, the value of M.s, which is left on the stack along with M, is really B.i and properly found right below where the reduction to B will later occur."
Example 5.19 [p. 335]

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

Last time we discussed the semantics of this statement.
S -> while ( 
  { L1 = new ();
    L2 = new();
    C.false = S.next;
    C.true = L2;
  }
  C )

  { S1.next = L1; }

S1

  { S.code = label || L1 || C.code ||
    label || L2 || S1.code; 
  }
Example 5.26 [p. 349]
Introducing marker non-terminals

S → while ( 
    M C )
N S1

{ S.code = label || L1 || C.code ||
    label || L2 || S1.code; }

M → ε

{ L1 = new ();
    L2 = new();
    C.false = S.next;
    C.true = L2;
}

N → ε

{ S1.next = L1; }
Example 5.26 [p. 349]
Introducing marker non-terminals

(put figure 5.37 on board)
Phases of a compiler

Intermediate Representation (IR): specification and generation

Figure 1.6, page 5 of text
Machine independent optimizations

IR → IR

Machine dependent optimizations

target 1

... target n