Semantic analysis of expressions & Statements

Prof. Vaibhavi Patel
Assistant Professor
Computer Science & Engineering

Content

INDEX

Semantic analysis

S-attributed definitions- It uses only synthesized attributes, it is called as S-attributed SDT.

- S-attributed SDT can be evaluated in bottom up order of the nodes of the parse tree.

L-attributed definitions- It uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

- Non-terminal can get values from its parent, child, and left sibling nodes.

SDD for expression grammar with synthesized attributes: S-attributed definition

Production	Semantic Rules
1) $L \rightarrow E$ **n**	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F \rightarrow$ **digit**	$F.val =$ **digit**.*lexval*

Expression statement

Assignment statement

- Non-terminal can get values from its child.
- In parse tree the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

SDD for expression grammar with inherited attributes: L-attributed definition

PRODUCTION	SEMANTIC RULE
$D \rightarrow TL$	$L.in := T.type$
$T \rightarrow$ int	$T.type := integer$
$T \rightarrow$ real	$T.type := real$
$L \rightarrow L_1$, id	$L_1.in := L.in$; $addtype(\text{id}.entry, L.in)$
$L \rightarrow$ id	$addtype(\text{id}.entry, L.in)$

Declarations of variables

- Synthesized: T.type
- Inherited: L.in

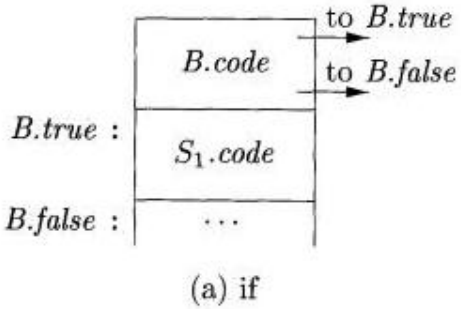
SDD for Control flow statements

- **Flow-of-Control Statements**
  - S -> if (B) S_1
  - S -> if (B) S_1 else s_2
  - S -> while (B) S_1

In these productions, nonterminal *B* represents a Boolean expression and non-terminal S represents a statement.

SDD for Control flow statements

PRODUCTION	*SEMANTIC RULES*
S -> if(B)S1	*B.true = newlabelQ* *B.false = Si.next = S.next* *S.code = B.code || labelQB .true) || Si.code*



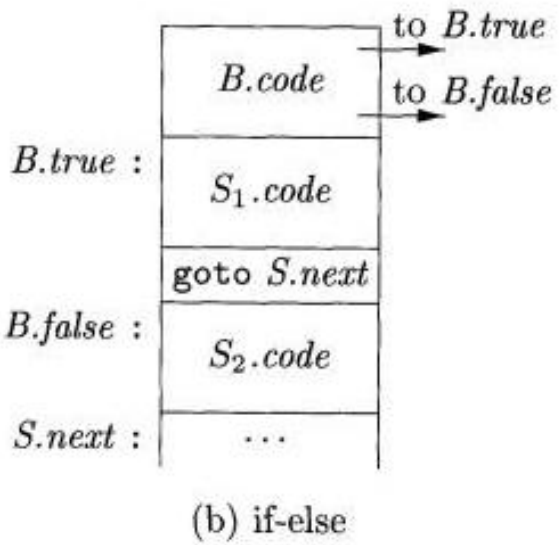
(a) if

- We assume that newlabelQ creates a new label each time it is call.
- Jumps to B.true within the code for B will go to the code for Si.
- Further, by setting B.false to S.next, we ensure that control will skip the code for Si if B evaluates to false.

SDD for Control flow statements

PRODUCTION	*SEMANTIC RULES*
s -> if (B) Si else S2	*B.true = newlabelQ* *B.false = newlabelQ* *Si.next = S2.next = S.next* *S.code = B.code* *|| label(B.true) \\ Si.code* *|| gen('goto' S.next)* *|| label(B.false) \\ S^.code*

SDD for Control flow statements



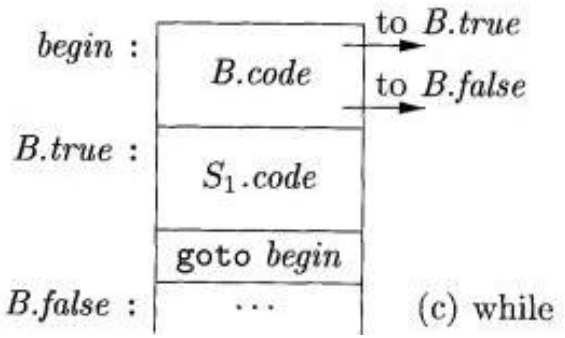
(b) if-else

- The code for the boolean expression *B* has jumps out of it to the first instruction of the code for Si if *B* is true, and to the first instruction of the code for S_2 if *B* is false.
- Further, control flows from both *Si* and S2 to the instruction immediately following the code for *S* — its label is
- given by the inherited attribute *S.next.*
- An explicit **goto** *S.next* appears after the code for Si to skip over the code for S2 .
- No goto is needed after S2 , since *Si.next* is the same as *S.next.*

SDD for Control flow statements

PRODUCTION	*SEMANTIC RULES*
S-> while (B) Si	*begin — newlabelQ* *B.true = newlabelQ* *B.false = S.next* *Si.next — begin* *S.code — label(begin) || B.code || label(B.true) \\ Si.code || geniigoto' &e#m)*

SDD for Control flow statements



- *Si* is formed from *B.code* and *Si.code.*
- We use a local variable *begin* to hold a new label attached to the first instruction for this while-statement, which is also the first instruction for *B.*
- We use a variable rather than an attribute, because *begin* is local to the semantic rules for this production.
- The inherited label *S.next* marks the instruction that control must flow to if *B* is false; hence, *B.false* is set to be *S.next.* A new label *B.true* is attached to the first instruction for Si; the code for *B* generates a jump to this label if *B* is true.
- After the code for Si we place the instruction **goto** *begin,* which causes a jump back to the beginning of the code for the boolean expression.
- Note that *Si.next* is set to this label *begin,* so jumps from within *Si.code* can go directly to *begin.*

SDD for Declarations of Variables & Functions

Purpose

- To enter identifiers (variables, arrays, functions) into the symbol table
- To assign type information
- To manage scope

SDD for Declarations of Variables & Functions

PRODUCTION	*SEMANTIC RULES*
D → T id ;	*id.entry.type = T.type* *Insert(id.entry) into current Symbol Table*
F → T id (ParamList) { S }	*id.entry.returnType = T.type* *id.entry.paramTypes = ParamList.types* *Create new scope for S*

SDD for Function Calls

PRODUCTION	SEMANTIC RULES
Call → id (ArgList)	*ArgList → E* *ArgList.types = [E.type]*
ArgList → ArgList , E	*$ArgList_1$ → $ArgList_2$, E* *$ArgList_1$.types = $ArgList_2$.types + [E.type]*
ArgList → E	*Call → id (ArgList)* *Check: id.entry.paramTypes == ArgList.types* *Call.type = id.entry.returnType*

SDD for Arrays

PRODUCTION	*SEMANTIC RULES*
D → T id [num] ;	*id.entry.type = array(T.type, num.value)* *Insert(id.entry) into Symbol Table*

SDD for Structures

PRODUCTION	*SEMANTIC RULES*
D → struct { FieldList } id ;	*id.entry.type = struct(FieldList.types)* *Insert(id.entry) into Symbol Table*

References

1. Compilers: Principles, Techniques, and Tools , by A.V. Aho, Monica Lam, Ravi Sethi, and J.D. Ullman, (2nd ed.), Addison-Wesley, 2007
2. brainkart.com
3. www.geeksforgeeks.org
4. www.tutorialspoint.com
5. javatpoint.com
6. wordpress.com

Parul® University | NAAC GRADE A++

https://paruluniversity.ac.in/