CS323 Lab 9

Yepang Liu

liuyp1@sustech.edu.cn

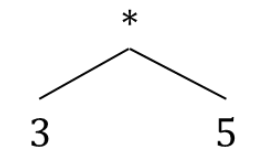
Outline

- Constructing Syntax tree
- The Structure of a Type

- Applications of Syntax-Directed Translation (Lab)
- Uses of SDTs (Lab)
- Implementing L-Attributed SDD's (Lab)
- Symbol Table Management

Construction of Syntax Tree

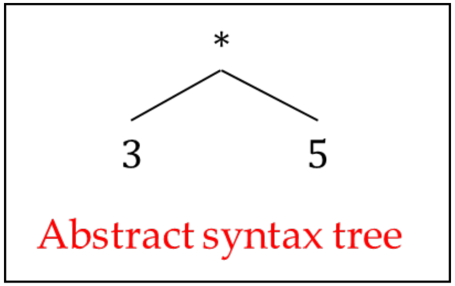
- Abstract syntax tree (or syntax tree for short) revisited:
  - Each interior node N represents a construct (corresponding to an operator)
  - The children of N represent the meaningful components of the construct represented by N (corresponding to operands)

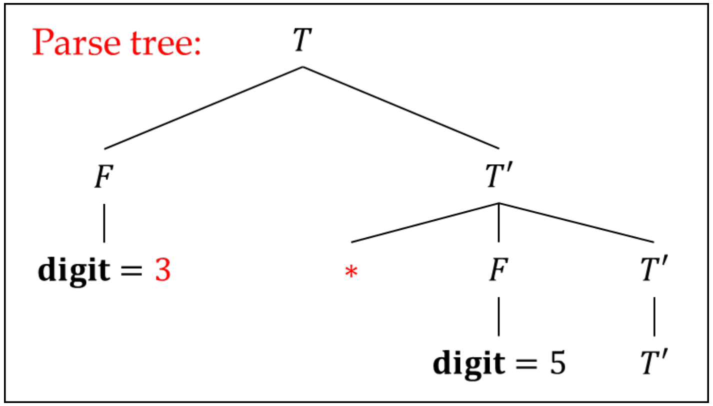


Abstract syntax tree

Construction of Syntax Tree

- Syntax tree vs. parse tree
  - In a syntax tree, interior nodes represent programming constructs, while in a parse tree, interior nodes represent nonterminals*
  - A parse tree is also called a *concrete syntax tree*, and the underlying grammar is called a *concrete syntax* for the language



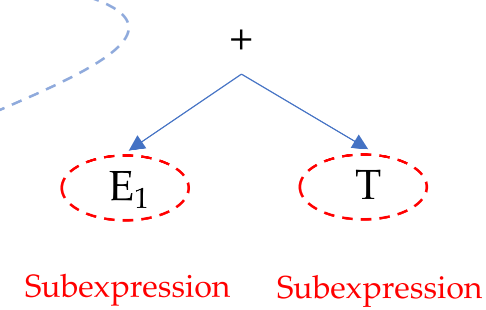


*Not all nonterminals represent programming constructs, e.g., those introduced to eliminate left recursions (T' in the earlier L-attributed SDD example)

Construction of Syntax Tree

- An S-attributed SDD for building syntax trees for simple expressions
  - Each node of the syntax tree is implemented as an **object** with a field *op*, representing the label of the node, and some additional fields
    - **Leaf node:** one additional field holding the lexical value
    - **Interior node:** # additional fields = # of children

	PRODUCTION	SEMANTIC RULES
1)	$E \to E_1 + T$	$\boxed{E.node = \textbf{new}\ Node('+', E_1.node, T.node)}$
2)	$E \to E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (E)$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

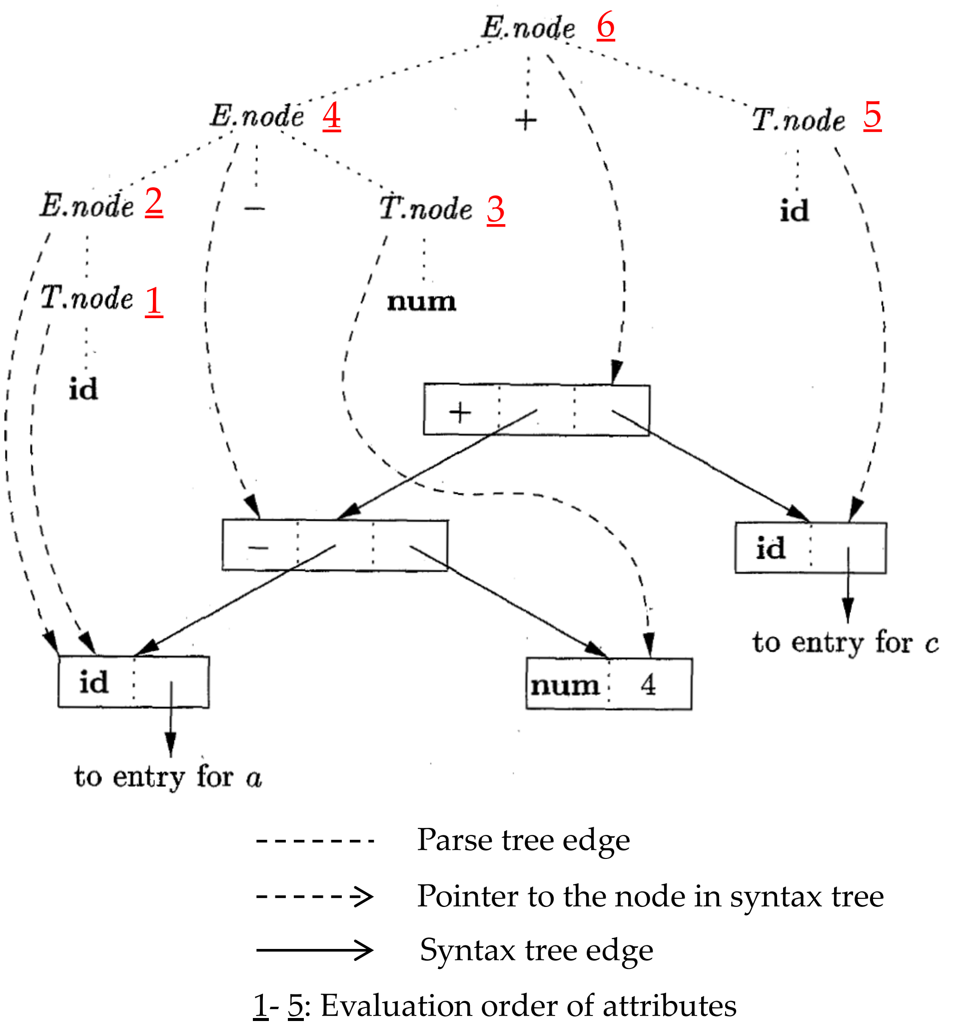


Construction of Syntax Tree

Input expression: $a - 4 + c$

Steps (object creations only; bottom-up evaluation):

1) $p_1 = \mathbf{new}\ \mathit{Leaf}(\mathbf{id}, \mathit{entry}\text{-}a);$
2) $p_2 = \mathbf{new}\ \mathit{Leaf}(\mathbf{num}, 4);$
3) $p_3 = \mathbf{new}\ \mathit{Node}('-', p_1, p_2);$
4) $p_4 = \mathbf{new}\ \mathit{Leaf}(\mathbf{id}, \mathit{entry}\text{-}c);$
5) $p_5 = \mathbf{new}\ \mathit{Node}('+', p_3, p_4);$



Outline

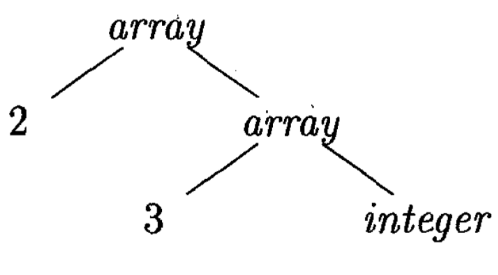
- Constructing Syntax tree
- The Structure of a Type

- Applications of Syntax-Directed Translation (Lab)
- Uses of SDTs (Lab)
- Implementing L-Attributed SDD’s (Lab)
- Symbol Table Management

Computing the Structure of a Type

```
int[2][3] a = ...;
```

What is the type of `a`?



elements

Element type

array(2, *array*(3, *integer*))

That is: array of 2 arrays of 3 integers

Computing the Structure of a Type

PRODUCTION
$T \rightarrow B\ C$
$B \rightarrow \textbf{int}$
$B \rightarrow \textbf{float}$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$
$C \rightarrow \epsilon$

The grammar generates type specifiers:

- `int`
- `float`

Basic types

- `int[2]`
- `int[2][3]`
- `int[4][5][6]`

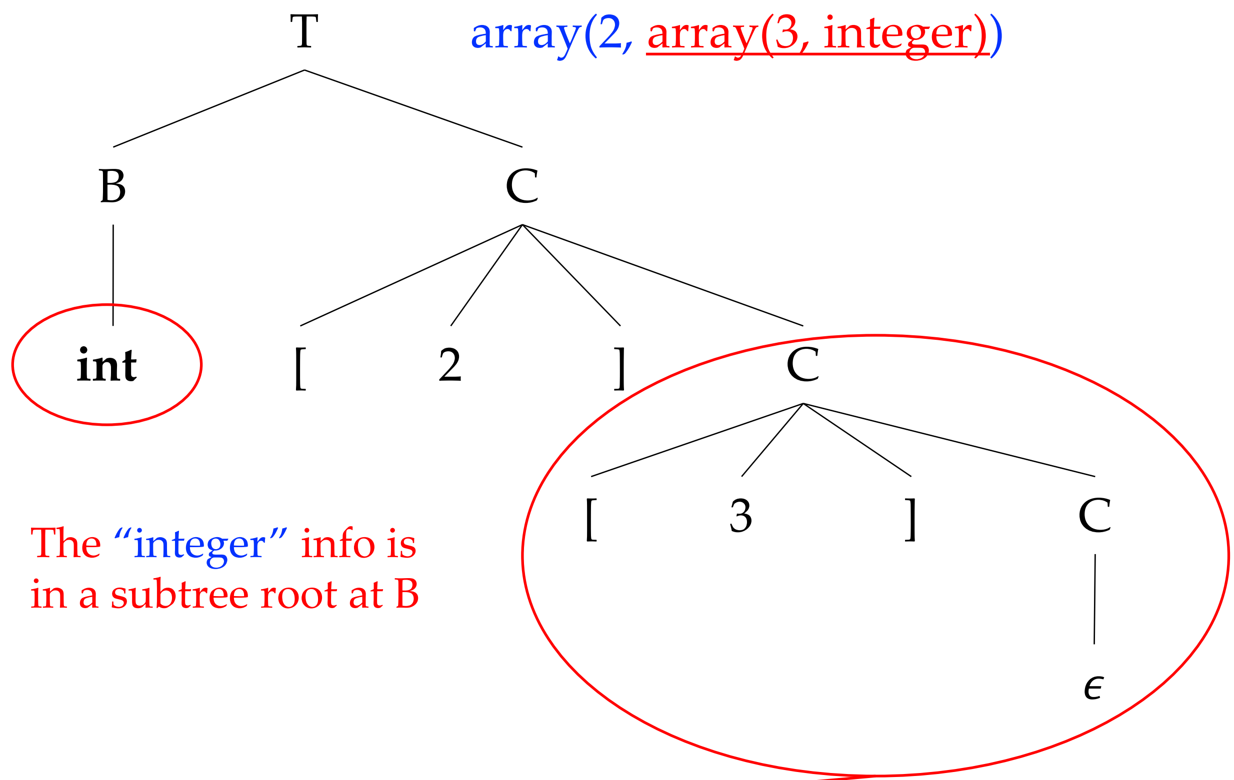
Array types

- ...

Computing the Structure of a Type

- `int[2][3]`

PRODUCTION
$T \rightarrow B\ C$
$B \rightarrow$ **int**
$B \rightarrow$ **float**
$C \rightarrow [$ **num** $]\ C_1$
$C \rightarrow \epsilon$



array(2, array(3, integer))

The "integer" info is in a subtree root at B

How can we obtain the type expression array(3, integer) from this subtree?

Computing the Structure of a Type

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow \textbf{int}$	$B.t = integer$
$B \rightarrow \textbf{float}$	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

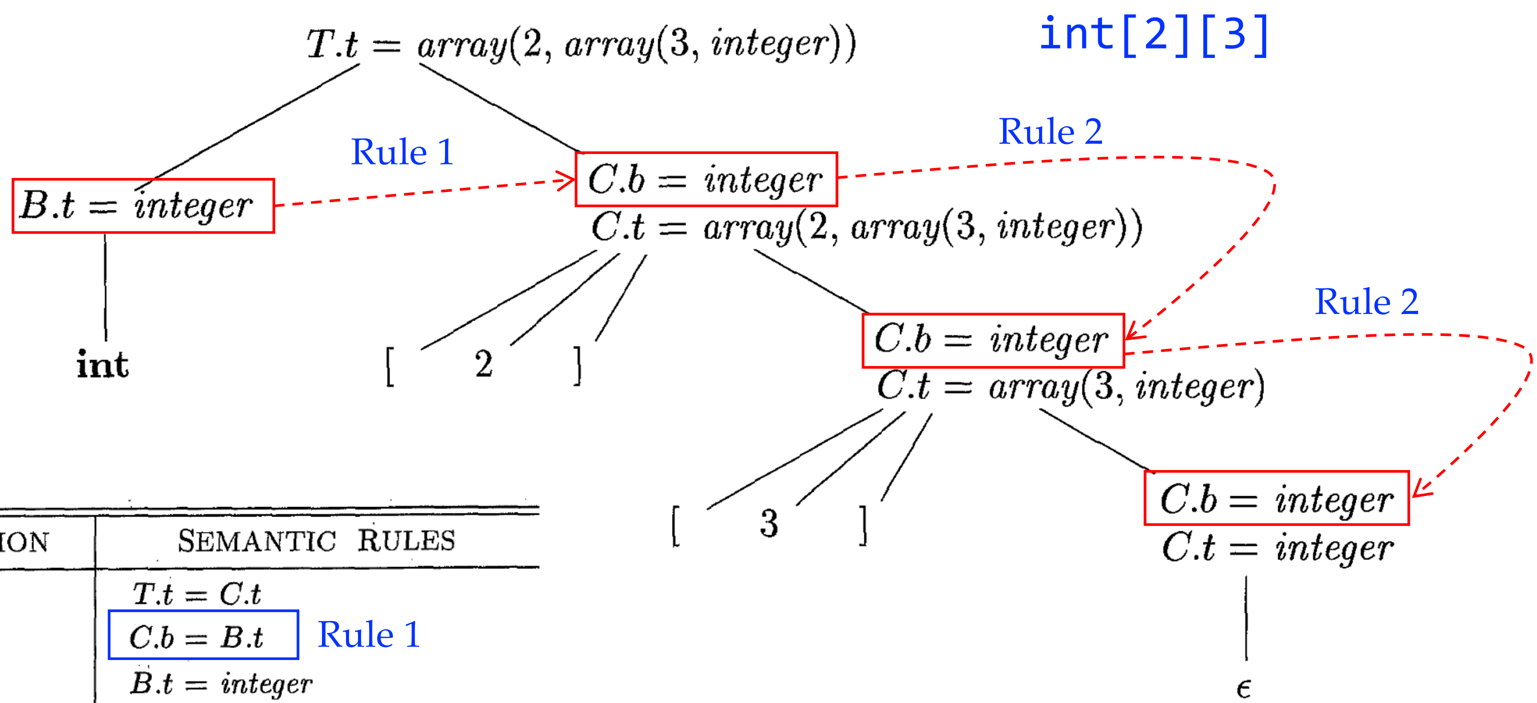
L-attributed SDD

Synthesized attribute t represents a type

Inherited attribute b passes the basic type down the parse tree

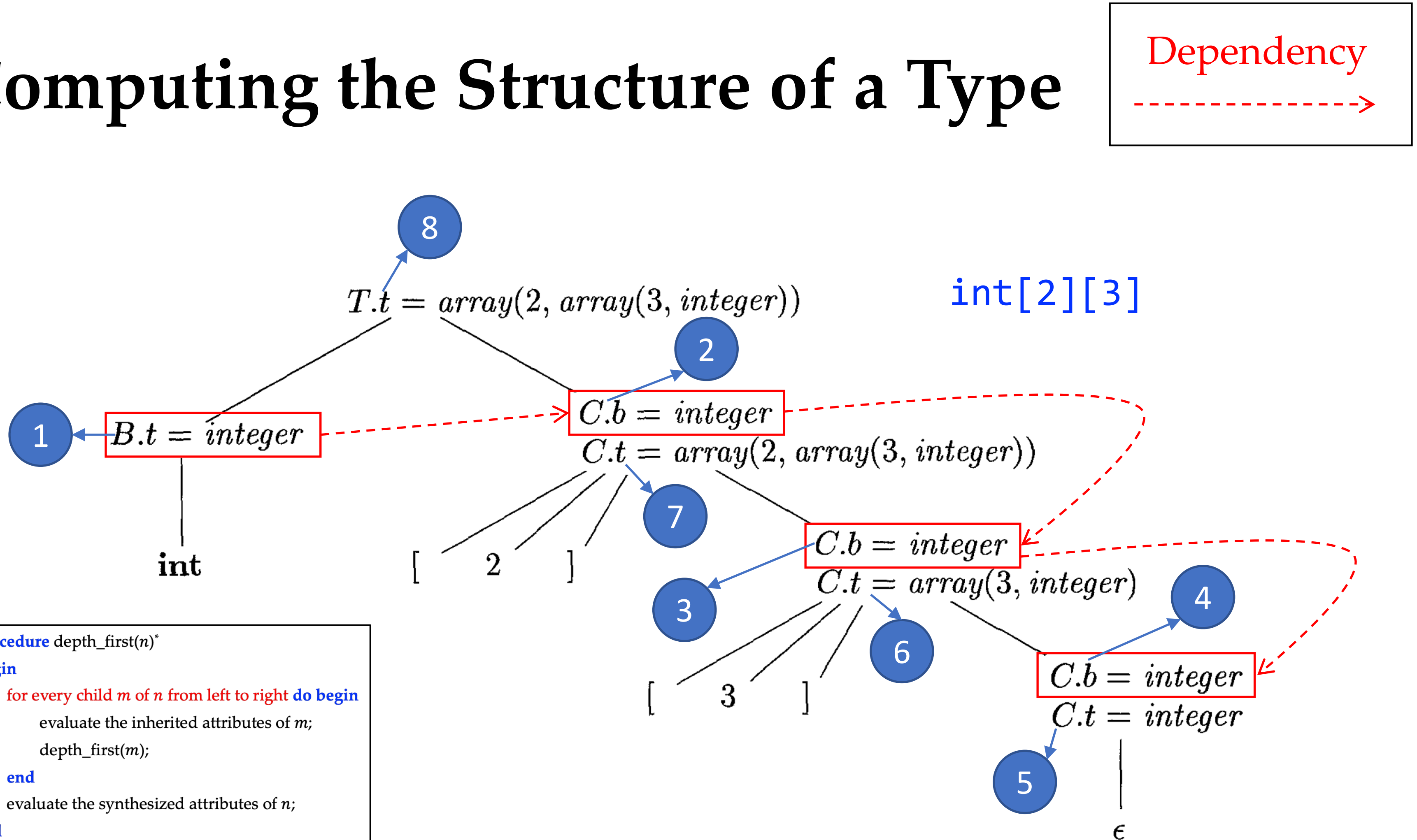
Computing the Structure of a Type

Dependency



PRODUCTION	SEMANTIC RULES	
$T \rightarrow B\ C$	$T.t = C.t$	
	$\boxed{C.b = B.t}$	Rule 1
$B \rightarrow$ **int**	$B.t = integer$	
$B \rightarrow$ **float**	$B.t = float$	
$C \rightarrow [$ **num** $]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$	
	$\boxed{C_1.b = C.b}$	Rule 2
$C \rightarrow \epsilon$	$C.t = C.b$	

Computing the Structure of a Type




```
procedure depth_first(n)*
begin
    for every child m of n from left to right do begin
        evaluate the inherited attributes of m;
        depth_first(m);
    end
    evaluate the synthesized attributes of n;
end
```

1 ... 8 : evaluation order (according to the algorithm on #23 of lecture notes)

Outline

- Constructing Syntax tree
- The Structure of a Type

- Applications of Syntax-Directed Translation (Lab)
- Uses of SDTs (Lab)
- Implementing L-Attributed SDD's (Lab)
- Symbol Table Management

Uses of SDT's

- We can use SDT's to implement two important classes of SDD's:
  - The underlying grammar is LR, and the SDD is S-attributed
  - The underlying grammar is LL, and the SDD is L-attributed

Postfix Translation Schemes

- If the grammar of an SDD is LR, and the SDD is S-attributed, then we can construct a *postfix SDT* (后缀SDT) to implement the SDD in bottom-up parsing
  - Semantic actions always appear at the end of productions (hence "postfix")

SDD

$L \to E\ \mathbf{n}$	$L.val = E.val$
$E \to E_1\ +\ T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1\ *\ F$	$T.val = T_1.val \times F.val$
$T \to F$	$T.val = F.val$
$F \to (\ E\ )$	$F.val = E.val$
$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

SDT

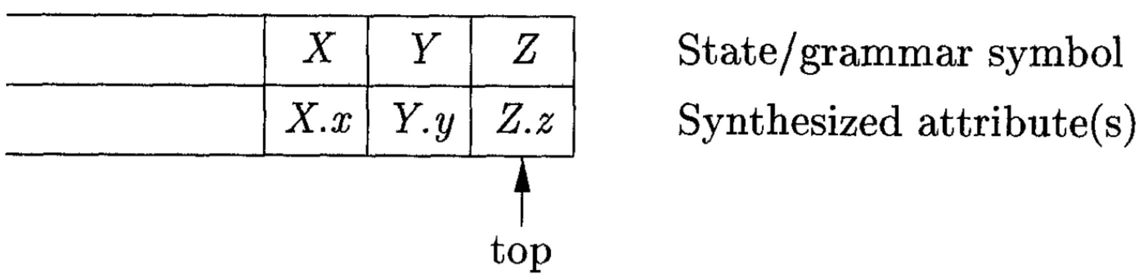
L	$\to$	$E\ \mathbf{n}$	$\{\ \text{print}(E.val);\ \}$
E	$\to$	$E_1 + T$	$\{\ E.val = E_1.val + T.val;\ \}$
E	$\to$	T	$\{\ E.val = T.val;\ \}$
T	$\to$	$T_1 * F$	$\{\ T.val = T_1.val \times F.val;\ \}$
T	$\to$	F	$\{\ T.val = F.val;\ \}$
F	$\to$	$(\ E\ )$	$\{\ F.val = E.val;\ \}$
F	$\to$	$\mathbf{digit}$	$\{\ F.val = \mathbf{digit}.lexval;\ \}$

This is possible because in bottom-up parsing, before reducing to a production head, the grammar symbols in the production body have been visited and their synthesized attributes have been computed (both non-terminals and terminals).

Parser-Stack Implementation of Postfix SDT's

- Postfix SDT's can be implemented during LR parsing by executing the actions when reductions occur

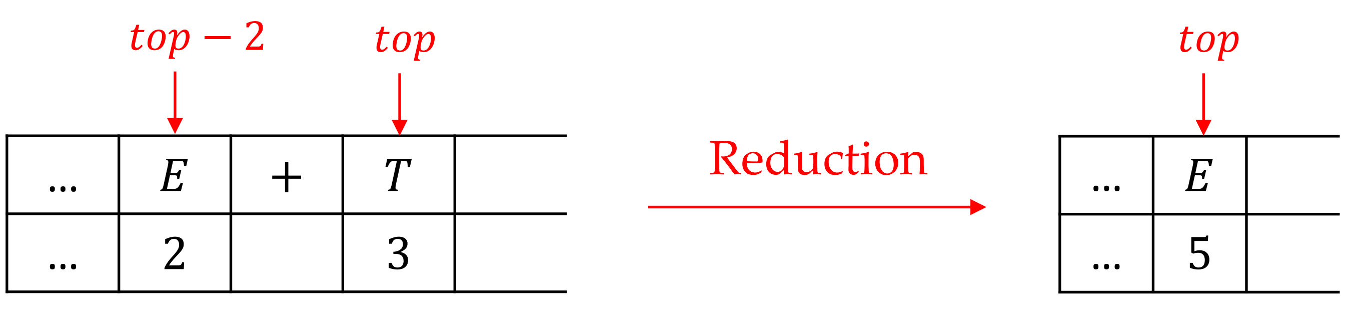
- The synthesized attributes can be placed along with the grammar symbols on the stack



If we do reduction using $A \rightarrow XYZ$, then the attributes of A can be calculated based on the attributes of X, Y, and Z, which are already on the stack.

The Calculator Example

PRODUCTION	ACTIONS
$L \to E\ \mathbf{n}$	$\{\ \text{print}(stack[top-1].val);$ $top = top - 1;\ \}$
$\boxed{E \to E_1 + T}$	$\boxed{\{\ stack[top-2].val = stack[top-2].val + stack[top].val;\ top = top - 2;\ \}}$
$E \to T$	
$T \to T_1 * F$	$\{\ stack[top-2].val = stack[top-2].val \times stack[top].val;$ $top = top - 2;\ \}$
$T \to F$	
$F \to (\ E\ )$	$\{\ stack[top-2].val = stack[top-1].val;$ $top = top - 2;\ \}$
$F \to \mathbf{digit}$	



Uses of SDT's

- We can use SDT's to implement two important classes of SDD's:
  - The underlying grammar is LR, and the SDD is S-attributed
  - The underlying grammar is LL, and the SDD is L-attributed

SDT's for L-Attributed SDD's

- L-attributed SDD's can be implemented during top-down parsing, if the underlying grammar is LL

- The way of turning an L-attributed SDD into an SDT is to place semantic actions at appropriate positions in the concerned production $A \rightarrow X_1 X_2 \ldots X_n$
  - Embed the action that computes the inherited attributes for a nonterminal X_i immediately before X_i in the production body
  - Place the actions that compute a synthesized attribute for the production head at the end of the production body

An L-Attributed SDD

- The SDD generates labels for the `while` loop

$$
\begin{array}{ll}
S \rightarrow \textbf{while} \; (\; C \;) \; S_1 & L1 = new(); \\
& L2 = new(); \\
& S_1.next = L1; \\
& C.false = S.next; \\
& C.true = L2; \\
& S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code
\end{array}
$$

L1

L2

```
while (condition) {
    doSomething();
}
```

Inherited attributes: $S.next$, $C.true$, $C.false$

Synthesized attribute: $S.code$

* There will be jump instructions with the labels as targets in $C.code$ and $S_1.code$.

Turning into an SDT

- Semantic actions:
  a) $L1 = new(); \ L2 = new();$
  b) $C.false = S.next; \ C.true = L2;$
  c) $S_1.next = L1;$
  d) $S.code = \cdots;$
- According to the rules of action placement:
  - b) should be placed before C, c) should be placed before S_1, and d) should be placed at the end of the production body
  - a) can be placed at the very beginning; there is no constraint

$$
\begin{array}{lll}
S \rightarrow \textbf{while} \ (& & \{ \ L1 = new(); \ L2 = new(); \ C.false = S.next; \ C.true = L2; \ \} \\
& C \ ) & \{ \ S_1.next = L1; \ \} \\
& S_1 & \{ \ S.code = \textbf{label} \ \| \ L1 \ \| \ C.code \ \| \ \textbf{label} \ \| \ L2 \ \| \ S_1.code; \ \}
\end{array}
$$

Outline

- Constructing Syntax tree
- The Structure of a Type

- Applications of Syntax-Directed Translation (Lab)
- Uses of SDTs (Lab)
- Implementing L-Attributed SDD's (Lab)
- Symbol Table Management

Translation During Recursive-Descent Parsing

- Many translation applications can be addressed using L-attributed SDD's. It is possible to extend a recursive-descent parser to implement L-attributed SDD's.
  - A recursive-decent parser has a function *A* for each nonterminal *A*

```
void A() {
    Choose an A-production, A → X1 X2 ··· Xk;
    for ( i = 1 to k ) {
        if ( Xi is a nonterminal )
            call procedure Xi();
        else if ( Xi equals the current input symbol a )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}
```

Translation During Recursive-Descent Parsing

- Extend a recursive-descent parser to implement L-attributed SDD's as follows:
  - A recursive-decent parser has a function *A* for each nonterminal *A*
  - Use the arguments of function *A* to **pass** *A*'s inherited attributes so that children nodes on the parse tree can use the attributes
  - **Return** the synthesized attributes of *A* when the function *A* completes so that parent node on the parse three can use the attributes
- With the above extension, in the body of the function *A*, we need to both parse and handle attributes

The While-Loop Example

$\boxed{S \rightarrow \textbf{while}\ (\ C\ )\ S_1}$

Save attributes in local variables

Pass inherited attributes
(the label of the statement after while)

```
string S(label next) {
    string Scode, Ccode; /* local variables holding code fragments */
    label L1, L2; /* the local labels */
    if ( current input == token while ) {
        advance input;
        check '(' is next on the input, and advance;
        L1 = new();
        L2 = new();
        Ccode = C(next, L2);
        check ')' is next on the input, and advance;
        Scode = S(L1);
        return("label" || L1 || Ccode || "label" || L2 || Scode);
    }
    else /* other statement types */
}
```

$C.false$ $C.true$

Pass inherited attributes when further handling other nonterminals

$S_1.next$ (the label of the condition evaluating statement)

Compute synthesized attributes and return

We mainly put code that handles attributes here, the code is not complete.

Outline

- Constructing Syntax tree
- The Structure of a Type

- Applications of Syntax-Directed Translation (Lab)
- Uses of SDTs (Lab)
- Implementing L-Attributed SDD's (Lab)
- Symbol Table Management

Symbol Table

- A *symbol table* maps an identifier (name) to its associated information
  - **identifier**: variable name, function name, user-defined type name (the name of the struct type in SPL), …
  - **information**: types, array dimension, struct members, initial values, …

Key: identifier → **Value:** the associated information

A symbol table is essentially a set of such key-value pairs

Symbol Table Operations

- **Symbol table operations during compilation**
  - **lookup**: check for variable existence, type definition, …
  - **insert**: when seeing function/variable/type declarations, …
  - **delete**: current scope finished, delete all identifiers inside (may not need this operation if only global scope is supported)

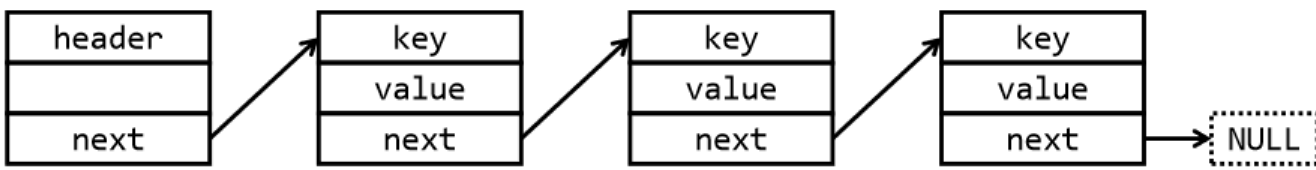
```
ExtDef -> Specifier ExtDecList        <- Handle global variables when reducing using this production
ExtDef -> Specifier SEMI              <- Handle user-defined types
ExtDef -> Specifier FunDec CompSt     <- Handle functions
Def -> Specifier DecList SEMI         <- Handle local variables
```

Symbol Table Implementation

- You are free to implement symbol table in terms of:
  - Stored information
    - Our suggestion: only store type information, including type info for variables, function return values, function parameters, and self-define data types
  - Possible choices of abstract data types:
    - linked list, hash table, binary search tree, …

Abstract Data Types

- **Linked list**



- **Lookup**: O(n) in worst case
- **Insert**: O(1) at head, O(n) at tail
- **Delete**: O(n) in worst case

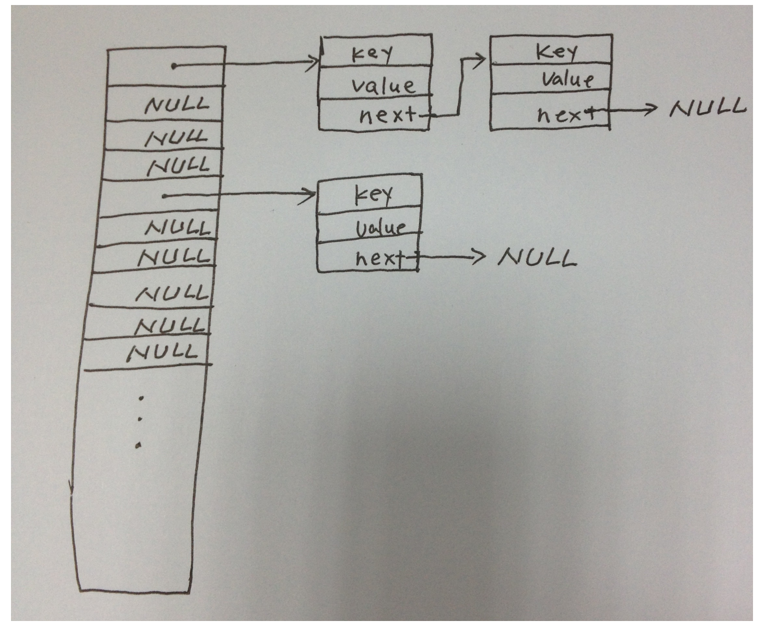
Abstract Data Types

- **Hash table**
  - Allocate a large consecutive space
  - Compress key to index (hash function)*
  - Most operations can be done in O(1)
  - Drawback: space consumption

* You may consider using https://en.wikipedia.org/wiki/PJW_hash_function

Abstract Data Types

- **Hash table conflicts**
  - When the hash functions maps multiple keys to the same index
  - **Solution 1**: Separate chaining (分离链接法)
  - **Solution 2**: Rehashing (再哈希法), which uses multiple hash functions and recomputes the hash value by an alternative hash function upon collisions



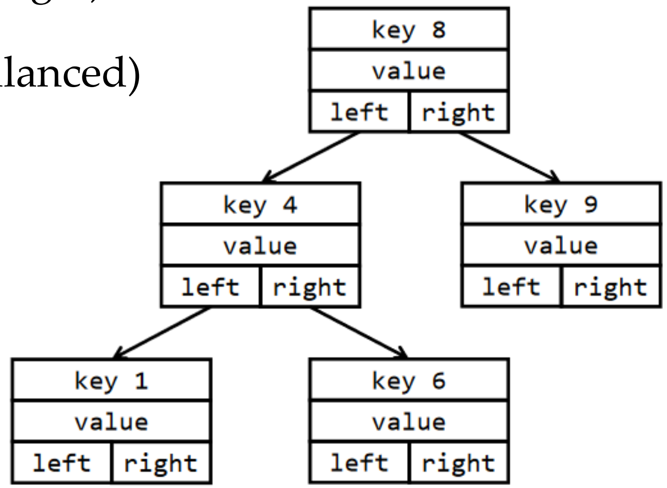
Separate chaining

Abstract Data Types

- **Binary search tree**
  - The key in each node is greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree
  - Ideally, the time complexity of operations: O(log n)
  - O(n) in worst case (when tree extremely imbalanced)
  - Balance strategies:*

    + AVL tree

    + Red-black tree



* https://www.javatpoint.com/red-black-tree-vs-avl-tree