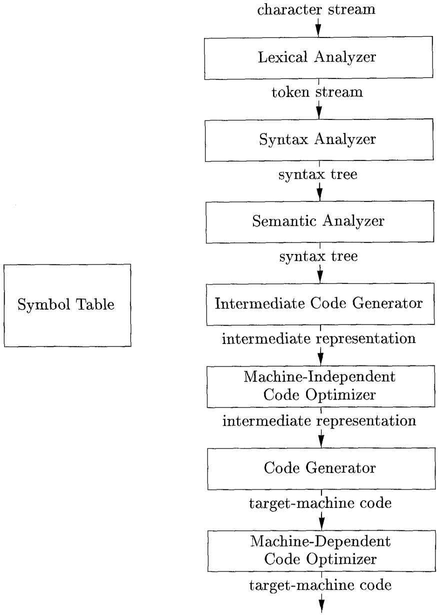
Partie 4

Semantic analysis

Structure of a compiler



Outline

1. Syntax-directed translation
2. Abstract syntax tree
3. Type and scope checking

Syntax-directed definition

- A general way to associate actions (i.e., programs) to production rules of a context-free grammar
- Used for carrying out most semantic analyses as well as code translation

- A syntax-directed definition associates:
  - With each grammar symbol, a set of attributes, and
  - With each production, a set of semantic rules for computing the values of the attributes associated with the symbols appearing in the production
- A grammar with attributes and semantic rules is called an attributed grammar
- A parse tree augmented with the attribute values at each node is called an annotated parse tree.

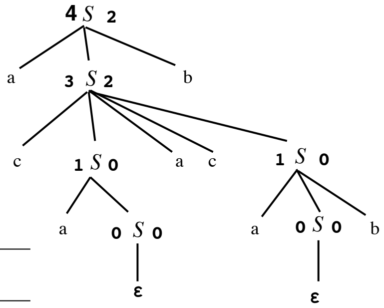
Example

Grammar:

$$S \rightarrow aSb|aS|cSacS|\epsilon$$

Semantic rules:

Production	Semantic rules
$S \rightarrow aS_1b$	$S.nba := S_1.nba + 1$ $S.nbc := S_1.nbc$
$S \rightarrow aS_1$	$S.nba := S_1.nba + 1$ $S.nbc := S_1.nbc$
$S \rightarrow cS_1acS_2$	$S.nba := S_1.nba + S_2.nba + 1$ $S.nbc := S_1.nbc + S_2.nbc + 2$
$S \rightarrow \epsilon$	$S.nba := 0$ $S.nbc := 0$
$S' \rightarrow S$	Final result is in $S.nba$ and $S.nbc$



(subscripts allow to distinguish different instances of the same symbol in a rule)

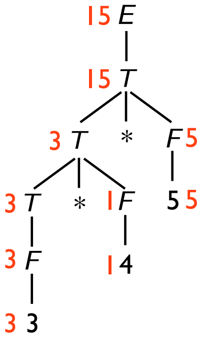
Attributes

- Two kinds of attributes
  - Synthesized: Attribute value for the LHS nonterminal is computed from the attribute values of the symbols at the RHS of the rule.
  - Inherited: Attribute value of a RHS nonterminal is computed from the attribute values of the LHS nonterminal and some other RHS nonterminals.
- Terminals can have synthesized attributes, computed by the lexer (e.g., $id.lexeme$), but no inherited attributes.

Example: synthesized attributes to evaluate expressions

Left-recursive expression grammar

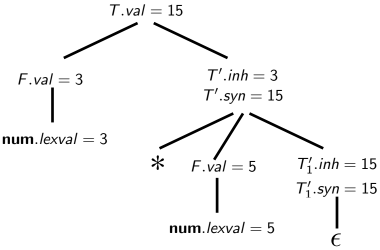
Production	Semantic rules
$L \rightarrow E$	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ **num**	$F.val =$ **num**$.lexval$



Example: inherited attributes to evaluate expressions

LL expression grammar

Production	Semantic rules
$T \rightarrow FT'$	$T'.inh = F.val$
	$T.val = T'.syn$
$T' \rightarrow *FT'_1$	$T'_1.inh = T'.inh \times F.val$
	$T'.syn = T'_1.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow$ **num**	$F.val = \mathbf{num}.lexval$



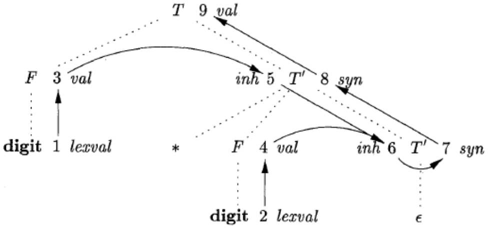
Evaluation order of SDD's

General case of synthesized and inherited attributes:

- Draw a dependency graph between attributes on the parse tree
- Find a topological order on the dependency graph (possible if and only if there are no directed cycle)
- If a topological order exists, it gives a working evaluation order. If not, it is impossible to evaluate the attributes

In practice, it is difficult to predict from a attributed grammar whether no parse tree will have cycles

Example:

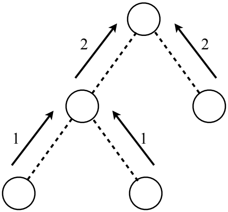


(Dragonbook)

Evaluation order of SDD's

Some important particular cases:

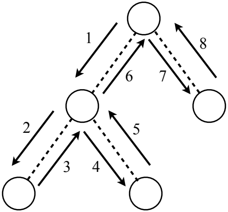
- A grammar with only synthesized attributes is called a S-attributed grammar.
- Attributes can be evaluated by a bottom-up (postorder) traversal of the parse tree



Evaluation order of SDD's

Some important particular cases:

- A syntax-directed definition is *L*-attributed if each attribute is either
  1. Synthesized
  2. Inherited "from the left": if the production is $A \rightarrow X_1 X_2 \dots X_n$, then the inherited attributes for X_j can depend only on
     - 2.1 Inherited attributes of A
     - 2.2 Any attribute among $X_1, \dots, X_{j-1}$ (symbols at the left of X_i
     - 2.3 Attributes of X_j (provided they are not causing cycles)
- To evaluate the attributes: do a depth first traversal evaluating inherited attributes on the way down and synthesized attributes on the way up (i.e., an Euler-tour traversal)



Translation of code

- Syntax-directed definitions can be used to translate code
- Example: translating expressions to post-fix notation

Production	Semantic rules
$L \rightarrow E$	$L.t = E.t$
$E \rightarrow E_1 + T$	$E.t = E_1.t \vert\vert T.t \vert\vert '+'$
$E \rightarrow E_1 - T$	$E.t = E_1.t \vert\vert T.t \vert\vert '-'$
$E \rightarrow T$	$E.t = T.t$
$T \rightarrow T_1 * F$	$T.t = T_1.t \vert\vert F.t \vert\vert '*'$
$T \rightarrow F$	$T.t = F.t$
$F \rightarrow (E)$	$F.t = E.t$
$F \rightarrow$ **num**	$F.t =$ **num**.*lexval*

Syntax-directed translation scheme

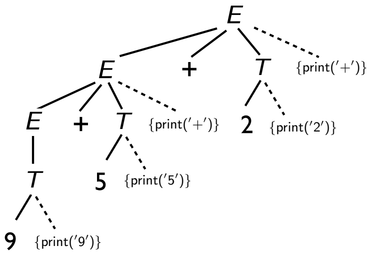
- The previous solution requires to manipulate strings (concatenate, create, store)
- An alternative is to use syntax-directed translation schemes.
- A syntax-directed translation scheme (SDT) is a context-free grammar with program fragments (called semantic actions) embedded within production bodies:

$$A \rightarrow \{R_0\}X_1\{R_1\}X_2 \ldots X_k\{R_k\}$$

- Actions are performed from left-to-right when the rules is used for a reduction
- Interesting for example to generate code incrementally

Example for code translation

Production	
$L \rightarrow E$	
$E \rightarrow E_1 + T$	$\{\text{print}('+')\}$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$\{\text{print}('*')\}$
$T \rightarrow F$	
$F \rightarrow (E)$	
$F \rightarrow$ **num**	$\{\text{print}(\textbf{num}.\textit{lexval})\}$



(Post-fix SDT as all actions are performed at the end of the productions)

Side-effects

- Semantic rules and actions in SDD and SDT's can have side-effects. E.g., for printing values or adding information into a table
- Needs to ensure that the evaluation order is compatible with side-effects
- Example: variable declaration in C

Production	Semantic rules	
$D \rightarrow TL$	$L.type = T.type$	(inherited)
$T \rightarrow \text{int}$	$T.type = \text{int}$	(synthesized)
$T \rightarrow \text{float}$	$T.type = \text{float}$	(synthesized)
$L \rightarrow L_1, \text{id}$	$L_1.type = L.type$	(inherited)
	$AddType(\mathbf{id}.entry, L.type)$	(synthesized, side effect)
$L \rightarrow \mathbf{id}$	$AddType(\mathbf{id}.entry, L.type)$	(synthesized, side effect)

- **id**.*entry* is an entry in the symbol table. *AddType* add type information about entry in the symbol table

Implementation of SDD's

Attributes can be computed after parsing:

- By explicitely traversing the parse or syntax tree in any order permitting the evaluation of the attributes
- Depth-first for S-attributed grammars or Euler tour for L-attributed grammar
- Advantage: does not depend on the order imposed by the syntax analysis
- Drawback: requires to build (and store in memory) the syntax tree

Evaluation after parsing of L-attributed grammar

For L-attribute grammars, the following recursive function will do the computation for inherited and synthesized attributes

```
ANALYSE(N, InheritedAttributes)
    if LEAF(N)
        return SynthesizedAttributes
    Attributes = InheritedAttributes
    for each child C of N, from left to right
        ChildAttributes = ANALYSE(C, Attributes)
        Attributes = Attributes ∪ ChildAttributes
    Execute semantic rules for the production at node N
    return SynthesizedAttributes
```

- Inherited attributes are passed as arguments and synthesized attributes are returned by recursive calls
- In practice, this is implemented as a big two-levels switch on nonterminals and then rules with this nonterminal at its LHS

Variations

- Instead of a giant switch, one could have separate routines for each nonterminal (as with recursive top-down parsing) and a switch on productions having this nonterminal as LHS (see examples later)
- Global variables can be used instead of parameters to pass inherited attributes by side-effects (with care)
- Can be easily adapted to use syntax-directed translation schemes (by interleaving child analysis and semantic actions)

Implementation of SDD's

Attributes can be computed directly during parsing:

- Attributes of a S-attributed grammar are easily computed during bottom-up parsing
- Attributes of a L-attributed grammar are easily computed during top-down parsing
- Attribute values can be stored on a stack (the same as the one for parsing or a different one)
- Advantage: one pass, does not require to store (or build) the syntax tree
- Drawback: the order of evaluation is constrained by the parser

Bottom-up parsing and S-attributed grammar

- Synthesized attributes are easily handled during bottom-up parsing. Handling inherited attributes is possible (for a LL-grammar) but more difficult.
- Example with only synthesized attributes (stored on a stack):

Production	Semantic rules	Stack actions
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$	$tmpT = \text{POP}()$
		$tmpE = \text{POP}()$
		$\text{PUSH}(tmpE + tempT)$
$E \rightarrow T$	$E.val = T.val$	
$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$	$tmpT = \text{POP}()$
		$tmpF = \text{POP}()$
		$\text{PUSH}(tmpT * tempF)$
$T \rightarrow F$	$T.val = F.val$	
$F \rightarrow (E)$	$F.val = E.val$	
$F \rightarrow$ **num**	$F.val =$ **num**$.lexval$	$\text{PUSH}($**num**$.lexval)$

(Parsing tables on slide 188)

Bottom-up parsing and S-attributed grammar

Stack	Input	Action	Attribute stack
$\$\boxed{0}$	$2*(10+3)\$$	s5	
$\$\boxed{0}\,2\boxed{5}$	$*(10+3)\$$	r6: $F \rightarrow$ **num**	2
$\$\boxed{0}F\boxed{3}$	$*(10+3)\$$	r4: $T \rightarrow F$	2
$\$\boxed{0}T\boxed{2}$	$*(10+3)\$$	s7	2
$\$\boxed{0}T\boxed{2}*\boxed{7}$	$(10+3)\$$	s4	2
$\$\boxed{0}T\boxed{2}*\boxed{7}(\boxed{4}$	$10+3)\$$	s5	2
$\$\boxed{0}T\boxed{2}*\boxed{7}(\boxed{4}\,10\boxed{5}$	$+3)\$$	r6: $F \rightarrow$ **num**	2 10
$\$\boxed{0}T\boxed{2}*\boxed{7}(\boxed{4}F\boxed{3}$	$+3)\$$	r4: $T \rightarrow F$	2 10
$\$\boxed{0}T\boxed{2}*\boxed{7}(\boxed{4}T\boxed{2}$	$+3)\$$	r2: $E \rightarrow T$	2 10
$\$\boxed{0}T\boxed{2}*\boxed{7}(\boxed{4}E\boxed{8}$	$+3)\$$	s6	2 10
$\$\boxed{0}T\boxed{2}*\boxed{7}(\boxed{4}E\boxed{8}+\boxed{6}$	$3)\$$	s5	2 10
$\$\boxed{0}T\boxed{2}*\boxed{7}(\boxed{4}E\boxed{8}+\boxed{6}\,3\boxed{5}$	$)\$$	r6: $F \rightarrow$ **num**	2 10 3
$\$\boxed{0}T\boxed{2}*\boxed{7}(\boxed{4}E\boxed{8}+\boxed{6}F\boxed{3}$	$)\$$	r4: $T \rightarrow F$	2 10 3
$\$\boxed{0}T\boxed{2}*\boxed{7}(\boxed{4}E\boxed{8}+\boxed{6}T\boxed{9}$	$)\$$	r1: $E \rightarrow E+T$	2 13
$\$\boxed{0}T\boxed{2}*\boxed{7}(\boxed{4}E\boxed{8}$	$)\$$	s11	2 13
$\$\boxed{0}T\boxed{2}*\boxed{7}(\boxed{4}E\boxed{8})\boxed{11}$	$\$$	r5: $F \rightarrow (E)$	2 13
$\$\boxed{0}T\boxed{2}*\boxed{7}F\boxed{10}$	$\$$	r3: $T \rightarrow T*F$	26
$\$\boxed{0}T\boxed{2}$	$\$$	r2: $E \rightarrow T$	26
$\$\boxed{0}E\boxed{1}$	$\$$	Accept	26

Top-down parsing of L-attributed grammar

- Recursive parser: the analysis scheme of slide 236 can be incorporated within the recursive functions of nonterminals
- Table-driven parser: this is also possible but less obvious.
- Example with only inherited attributes (stored on a stack):

Production	Semantic rules	Stack actions
$S' \rightarrow S$	$S.nb = 0$	$\text{PUSH}(0)$
$S \rightarrow (S_1)S_2$	$S_1.nb = S.nb + 1$ $S_2.nb = S.nb$	$\text{PUSH}(\text{TOP}() + 1)$
$S \rightarrow \epsilon$	$\text{PRINT}(S.nb)$	$\text{PRINT}(\text{POP}())$

(print the depth of nested parentheses)

Parsing table:

	(	)	\$
S'	$S' \rightarrow S$		$S' \rightarrow S$
S	$S \rightarrow (S)S$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$

Top-down parsing of L-attributed grammar

Stack	Input	Attribute stack	Output
$S'\$$	(()(()))()	0	
$S\$$	(()(()))()	0 1	
$(S)S\$$	(()(()))()	0 1 2	
$S)S\$$	()(()))()	0 1 2	
$(S)S)S\$$	()(()))()	0 1	2
$S)S)S\$$	)(()))()	0 1	
$)S)S\$$	)(()))()	0 1 2	
$S)S\$$	(()))()	0 1 2	
$(S)S)S\$$	(()))()	0 1 2 3	
$S)S)S\$$	()))()	0 1 2 3	
$(S)S)S)S\$$	()))()	0 1 2	3
$S)S)S)S\$$	)))()	0 1 2	
$)S)S)S\$$	)))()	0 1	2
$S)S)S\$$	))()	0 1	
$)S)S\$$	))()	0	1
$S)S\$$	)()	0	
$)S\$$	)()	0 1	
$S\$$	()	0 1	
$(S)S\$$	()	0	1
$S)S\$$	)	0	
$)S\$$	)		0
$S\$$			
$\$$			

Comments

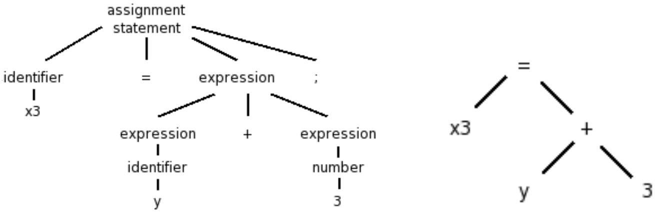
- It is possible to transform a grammar with synthesized and inherited attributes into a grammar with only synthesized attributes
- It is usually easier to define semantic rules/actions on the original (ambiguous) grammar, rather than the transformed one
- There are techniques to transform a grammar with semantic actions (see reference books for details)

Applications of SDD's

SDD can be used at several places during compilation:

- Building the syntax tree from the parse tree
- Various static semantic checking (type, scope, etc.)
- Code generation
- Building an intepreter
- ...

Abstract syntax tree

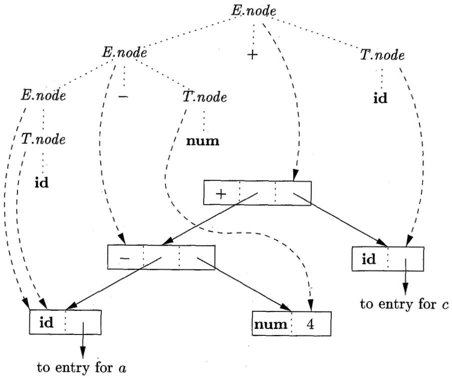


- The abstract syntax tree is often used as a basis for other semantic analysis or as an intermediate representation
- When the grammar has been modified for parsing, the syntax tree is a more natural representation than the parse tree
- The abstract syntax tree can be constructed using SDD (see next slides)
- Another SDD can then be defined on the syntax tree to perform semantic checking or generate another intermediate code (directed by the syntax tree and not the parse tree)

Generating an abstract syntax tree

For the left-recursive expression grammar:

Production	Semantic rules
$E \rightarrow E_1 + T$	$E.node = \textbf{new}\,Node('+', E_1.node, T.node)$
$E \rightarrow E_1 - T$	$E.node = \textbf{new}\,Node('-', E_1.node, T.node)$
$E \rightarrow T$	$E.node = T.node$
$T \rightarrow (E)$	$T.node = E.node$
$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\,Leaf(\textbf{id}, \textbf{id}.entry)$
$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\,Leaf(\textbf{num}, \textbf{num}.entry)$

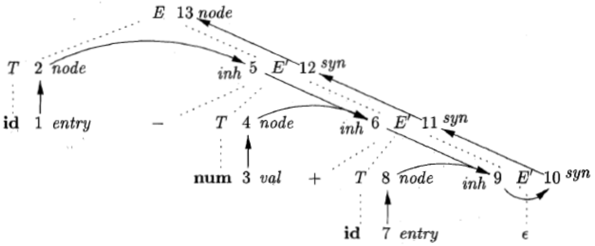


(Dragonbook)

Generating an abstract syntax tree

For the LL transformed expression grammar:

Production	Semantic rules
$E \rightarrow TE'$	$E.node = E'.syn$; $E'.inh = T.node$
$E \rightarrow +TE'_1$	$E'_1.inh = \textbf{new}Node('+', E'.inh, T.node)$; $E'.syn = E'_1.syn$
$E \rightarrow -TE'_1$	$E'_1.inh = \textbf{new}Node('-', E'.inh, T.node)$; $E'.syn = E'_1.syn$
$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
$E \rightarrow T$	$E.node = T.node$
$T \rightarrow (E)$	$T.node = E.node$
$T \rightarrow \textbf{id}$	$T.node = \textbf{new}Leaf(\textbf{id}, \textbf{id}.entry)$
$T \rightarrow \textbf{num}$	$T.node = \textbf{new}Leaf(\textbf{num}, \textbf{num}.entry)$



(Dragonbook)

Type and scope checking

- Static checkings:
  - All checkings done at compilation time (versus dynamic checkings done at run time)
  - Allow to catch errors as soon as possible and ensure that the program can be compiled
- Two important checkings:
  - Scope checking: checks that all variables and functions used within a given scope have been correctly declared
  - Type checking: ensures that an operator or function is applied to the correct number of arguments of the correct types
- These two checks are based on information stored in a symbol table

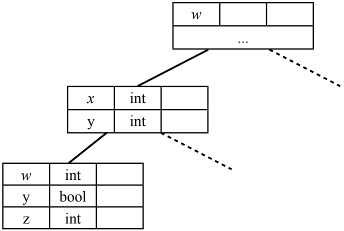
Scope

```
{
   int x = 1;
   int y = 2;
   {
      double x = 3.1416;
      y += (int)x;
   }
   y += x;
}
```

- Most languages offer some sort of control for scopes, constraining the visibility of an identifier to some subsection of the program
- A scope is typically a section of program text enclosed by basic program delimiters, e.g., {} in C, begin-end in Pascal.
- Many languages allow nested scopes, i.e., scopes within scopes. The current scope (at some program position) is the innermost scope.
- Global variables and functions are available everywhere
- Determining if an identifier encountered in a program is accessible at that point is called Scope checking.

Symbol table

```
{ int x; int y;
   {  int w; bool y;
        ..w..; ..x..; ..y..; ..z..;
   }
   ..w..; ..x..; ..y..;
}
```



- The compiler keeps track of names and their binding using a symbol table (also called an environment)
- A symbol table must implement the following operations:
  - Create an empty table
  - Add a binding between a name and some information
  - Look up a name and retrieve its information
  - Enter a new scope
  - Exit a scope (and reestablish the symbol table in its state before entering the scope)

Symbol table

- To manage scopes, one can use a persistent or an imperative data structure
- A persistent data structure is a data structure which always preserves the previous version of itself when it is modified
- Example: lists in functional languages such as Scheme
  - Binding: insert the binding at the front of the list, lookup: search the list from head to tail
  - Entering a scope: save the current list, exiting: recalling the old list
- A non persistent implementation: with a stack
  - Binding: push the binding on top of the stack, lookup: search the stack from top to bottom
  - Entering a scope: push a marker on the top of the stack, exiting: pop all bindings from the stack until a marker is found, which is also popped
  - This approach destroys the symbol table when exiting the scope (problematic in some cases)

More efficient data structures

- Search in list or stack is $O(n)$ for n symbols in the table
- One can used more efficient data structures like hash-tables or binary search trees
- Scopes can then be handled in several ways:
  - Create a new symbol table for each scope and use a stack or a linked list to link them
  - Use one big symbol table for all scopes:
    - Each scope receives a number
    - All variables defined within a scope are stored with their scope number
    - Exiting a scope: removing all variables with the current scope number
  - There exist persistent hash-tables

Types

- Type checking is verifying that each operation executed in a program respects the type system of the language, i.e., that all operands in any expression are of appropriate types and number
- Static typing if checking is done at compilation-time (e.g., C, Java, C++)
- Dynamic typing if checking is done at run-time (e.g., Scheme, Javascript).
- Implicit type conversion, or coercion, is when a compiler finds a type error and change the type of the variable into the appropriate one (e.g., integer→float)

Principle of type checking

- Identify the types of the language and the language constructs that have types associated with them
- Associate a type attribute to these constructs and semantic rules to compute them and to check that the typing system is respected
- Needs to store identifier types in the symbol table
- One can use two separate tables, one for the variable names and one for the function names
- Function types is determined by the types (and number) of arguments and return type. E.g., $(int, int) \rightarrow int$
- Type checking can not be dissociated from scope and other semantic checking

Illustration

We will use the following source grammar to illustrate type checking (and code generation next week)

$$
\begin{array}{lcl}
Program & \rightarrow & Funs \\
\\
Funs & \rightarrow & Fun \\
Funs & \rightarrow & Fun\ Funs \\
\\
Fun & \rightarrow & TypeId\ (\ TypeIds\ ) = Exp \\
\\
TypeId & \rightarrow & \texttt{int}\ \mathbf{id} \\
TypeId & \rightarrow & \texttt{bool}\ \mathbf{id} \\
\\
TypeIds & \rightarrow & TypeId \\
TypeIds & \rightarrow & TypeId\ ,\ TypeIds
\end{array}
$$

$$
\begin{array}{lcl}
Exp & \rightarrow & \mathbf{num} \\
Exp & \rightarrow & \mathbf{id} \\
Exp & \rightarrow & Exp + Exp \\
Exp & \rightarrow & Exp = Exp \\
Exp & \rightarrow & \texttt{if}\ Exp\ \texttt{then}\ Exp\ \texttt{else}\ Exp \\
Exp & \rightarrow & \mathbf{id}\ (\ Exps\ ) \\
Exp & \rightarrow & \texttt{let}\ \mathbf{id} = Exp\ \texttt{in}\ Exp \\
\\
Exps & \rightarrow & Exp \\
Exps & \rightarrow & Exp\ ,\ Exps
\end{array}
$$

(see chapter 5 and 6 of (Mogensen, 2010) for full details)

Implementation on the syntax tree: expressions

Type checking of expressions:

inherited attributes

$Check_{Exp}(Exp, vtable, ftable) = $ `case` Exp `of`	
num	`int`
id	$t = lookup(vtable, getname(\mathbf{id}))$ $if\ \ t = unbound$ $then$ **error**(); `int` $else\ \ t$
$Exp_1 + Exp_2$	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ $if\ \ t_1 =$ `int` $and\ t_2 =$ `int` $then$ `int` $else$ **error**(); `int`

filled in by lexer

scope checking

error recovery

type checking

synthesized attribute

Follows the implementation of slide 237 with one function per nonterminal, with a switch on production rules

Implementation on the syntax tree: function calls

$Check_{Exp}(Exp, vtable, ftable) =$ `case` Exp `of`	
id ($Exps$)	$t = lookup(ftable, getname(\mathbf{id}))$ $if\ \ t = unbound$ $then$ **error**(); `int` $else$ $((t_1, \ldots, t_n) \to t_0) = t$ $[t'_1, \ldots, t'_m] = Check_{Exps}(Exps, vtable, ftable)$ $if\ \ m = n\ and\ \ t_1 = t'_1, \ldots, t_n = t'_n$ $then\ \ t_0$ $else$ **error**(); t_0

filled in by lexer

scope checking

checking function arguments

$Check_{Exps}(Exps, vtable, ftable) =$ `case` $Exps$ `of`	
Exp	$[Check_{Exp}(Exp, vtable, ftable)]$
Exp , $Exps$	$Check_{Exp}(Exp, vtable, ftable)$ $:: Check_{Exps}(Exps, vtable, ftable)$

≈cons

Implementation on the syntax tree: variable declaration

$Check_{Exp}(Exp, vtable, ftable) = \texttt{case}\ Exp\ \texttt{of}$	
$\texttt{let}\ \mathbf{id} = Exp_1$ $\texttt{in}\ Exp_2$	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, getname(\mathbf{id}), t_1)$ $Check_{Exp}(Exp_2, vtable', ftable)$

create a new scope

- Create a new symbol table *vtable′* with the new binding
- Pass it as an argument for the evaluation of Exp_2 (*right* child)

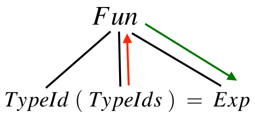
Implementation on the syntax tree: function declaration

synthesized attribute

$Check_{Fun}(Fun, ftable) =$ `case` Fun `of`	
$TypeId\ (\ TypeIds\ )\ =\ Exp$	$(f, t_0) = Get_{TypeId}(TypeId)$ $vtable = Check_{TypeIds}(TypeIds)$ $t_1 = Check_{Exp}(Exp, vtable, ftable)$ $if\ \ t_0 \neq t_1$ $then$ **error**()

inherited attributes

$Get_{TypeId}(TypeId) =$ `case` $TypeId$ `of`	
`int` **id**	$(getname($**id**$),$ `int`$)$
`bool` **id**	$(getname($**id**$),$ `bool`$)$



$Check_{TypeIds}(TypeIds) =$ `case` $TypeIds$ `of`	
$TypeId$	$(x, t) = Get_{TypeId}(TypeId)$ $bind(emptytable, x, t)$
$TypeId\ ,\ TypeIds$	$(x, t) = Get_{TypeId}(TypeId)$ $vtable = Check_{TypeIds}(TypeIds)$ $if\ \ lookup(vtable, x) = unbound$ $then\ \ bind(vtable, x, t)$ $else$ **error**(); $vtable$

Create a symbol table with arguments

Implementation on the syntax tree: program

$Check_{Program}(Program) = \texttt{case}\ Program\ \texttt{of}$	
$Funs$	$ftable = Get_{Funs}(Funs)$ $Check_{Funs}(Funs, ftable)$ $if\ \ lookup(ftable, \texttt{main}) \neq (\texttt{int}) \rightarrow \texttt{int}$ $then\ \ \mathbf{error}()$

Collect all function definitions in a symbol table (to allow mutual recursion)

Language semantic requires a main function

$Check_{Funs}(Funs, ftable) = \texttt{case}\ Funs\ \texttt{of}$	
Fun	$Check_{Fun}(Fun, ftable)$
$Fun\ Funs$	$Check_{Fun}(Fun, ftable)$ $Check_{Funs}(Funs, ftable)$

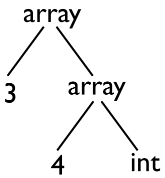
- Needs two passes over the function definitions to allow mutual recursion
- See (Mogensen, 2010) for Get_{Funs} (similar as $Check_{Funs}$)

More on types

- Compound types are represented by trees (constructed by a SDD)
- Example: array declarations in C

Production	Semantic rules
$T \rightarrow BC$	$T.t = C.t$; $C.b = B.t$
$B \rightarrow$ int	$B.t =$int
$B \rightarrow$ float	$B.t =$float
$C \rightarrow [$ NUM $]C_1$	$C.t = array(\text{NUM}.val, C_1.t)$
$C \rightarrow \epsilon$	$C.t = C.b$

`int [3][4]`



- Compound types are compared by comparing their trees

More on types

- Type coercion:
  - The compiler supply implicit conversions of types
  - Define a hierarchy of types and find for two operands the least upper bound (LUB) in the hierarchy
  - Convert both operands to the LUB type
- Overloading:
  - An operator accepting different types (e.g., $=$ in our source language)
  - Type must be defined at translation
- Polymorphism: functions defined over a large class of similar types
- Implicit types: some languages (like ML or Haskell) do not require to explicit declare type of functions or variables. Types are automatically inferred at compile time.