Compiler Design

Samit Biswas

samit@cs.iiests.ac.in

Department of Computer Science and Technology,
Indian Institute of Engineering Science and Technology, Shibpur

September 27, 2018

Syntax Directed Translation

Construction of Syntax tree

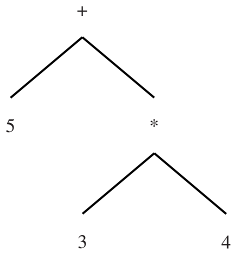
Syntax - Tree

- an intermediate representation of the **compiler's input.**
- A condensed form of parse tree.
- Syntax tree shows the syntactic structure of the programme while omitting the irrelevant details.
- Operators or keywords are associated with the interior nodes.

Syntax directed translation can be based on syntax tree as well as parse tree.

Syntax - Tree Example

5 + 3 * 4



- Leaves: identifiers or constants.
- Internal nodes: Labelled with operators.
- Children of a node are its operands.

Constructing Syntax trees for an Expression:

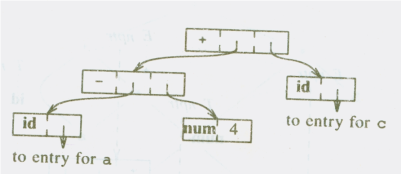
- Each node can be implemented as a record with several fields.
- Operator node: one field identifies the operator (called label of the node) and remaining fields contain pointers to operands.
- The nodes may also contain fields to hold the values (pointers to values) of attributes attached to the nodes.
- Functions used to create nodes of syntax tree for expressions with binary operator are given below
  - **mknode**(op, left, right)
  - **mkleaf**(id, entry)
  - **mkleaf** (num, val)

  Each function returns a pointer to a newly created node.

Example

a - 4 + c

1. $p_1 = mkleaf(id, entrya);$
2. $p_2 = mkleaf(num, 4);$
3. $p_3 = mknode('-', p_1, p_2);$
4. $p_4 = mkleaf(id, entryc);$
5. $p_5 = mknode('+', p_3, p_4);$

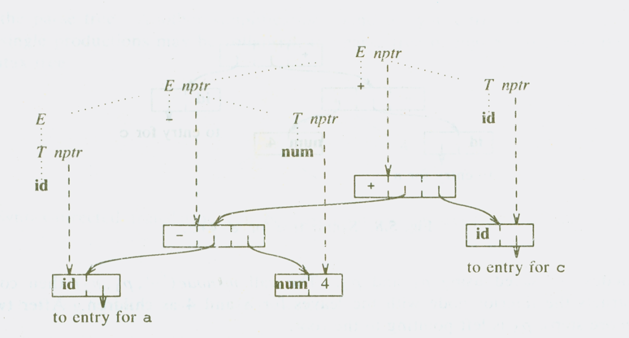


Syntax-Directed definition for Constructing Syntax Trees

Table: Syntax-directed definition for constructing a syntax tree.

Production	Semantic Rules
$E \rightarrow E_1 + T$	$E.nptr = mknode('+', E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr = mknode('-', E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow (E)$	$T.nptr = E.nptr$
$T \rightarrow id$	$T.nptr = mkleaf(id, id.entry)$
$T \rightarrow num$	$T.nptr = mkleaf(num, num.val)$

Example: Construction of a Syntax-tree for a-4+c



Directed Acyclic Graphs for Expression

Bottom up Evaluation of S - Attributed Definitions

- A translator for an S-attributed definition can often be implemented with the help of an LR parser.
- From an S-attributed definition the parser generator can construct a translator that evaluates attributes as it parses the input.
- We put the values of the synthesized attributes of the grammar symbols a stack that has extra fields to hold the values of attributes.
-

Table: Implementation of a Calculator with an LR parser

Production	Code Fragment
$L \rightarrow En$	print(val[top]);
$E \rightarrow E_1 + T$	$val[ntop] = val[top-2] + val[top]$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$val[ntop] = val[top-2] * val[top]$
$T \rightarrow F$	
$F \rightarrow (E)$	$val[ntop] = val[top-1]$
$F \rightarrow digit$	

Table: Moves made by translator on Input 3*5+4n

Input	State	val	Production Used
3*5+4n	–	–	

Table: Moves made by translator on Input 3*5+4n

Input	State	val	Production Used
3*5+4n	–	–	
*5+4n	3	3	

Table: Moves made by translator on Input 3*5+4n

Input	State	val	Production Used
3*5+4n	–	–	
*5+4n	3	3	
*5+4n	F	3	$F \rightarrow digit$

Table: Moves made by translator on Input 3*5+4n

Input	State	val	Production Used
3*5+4n	–	–	
*5+4n	3	3	
*5+4n	F	3	$F \rightarrow digit$
*5+4n	T	3	$T \rightarrow F$

Table: Moves made by translator on Input 3*5+4n

Input	State	val	Production Used
3*5+4n	–	–	
*5+4n	3	3	
*5+4n	F	3	$F \rightarrow digit$
*5+4n	T	3	$T \rightarrow F$
5+4n	T *	3 _	

Table: Moves made by translator on Input 3*5+4n

Input	State	val	Production Used
3*5+4n	–	–	
*5+4n	3	3	
*5+4n	F	3	$F \rightarrow digit$
*5+4n	T	3	$T \rightarrow F$
5+4n	T *	3 _	
+4n	T * 5	3 _ 5	

Table: Moves made by translator on Input 3*5+4n

Input	State	val	Production Used
3*5+4n	–	–	
*5+4n	3	3	
*5+4n	F	3	$F \rightarrow digit$
*5+4n	T	3	$T \rightarrow F$
5+4n	T *	3 _	
+4n	T * 5	3 _ 5	
+4n	F * F	3 _ 5	$F \rightarrow digit$
+4n	T	15	$T \rightarrow T * F$
+4n	E	15	$E \rightarrow T$
4n	E +	15 _	
n	E + 4	15 _ 4	
n	E + F	15 _ 4	$F \rightarrow digit$
n	E + T	15 _ 4	$T \rightarrow F$
n	E	19	$E \rightarrow E + T$
	E n	19 _	
	L	19	$L \rightarrow En$

Bottom-Up Evaluation of Inherited Attributes

Type Checking

Type Checking

- Type checking is the process of verifying that each operation executed in a program respects the type system of the language.
- This generally means that all operands in any expression are of appropriate types and numbers.
- Mostly what we do in semantic analysis phase is type checking.

Designing a Type Checker

When designing a Type Checker for a compiler here's the process:

- Identify the types that are available in the language.

Designing a Type Checker

When designing a Type Checker for a compiler here's the process:

- Identify the types that are available in the language.
- Identify the language construct that have types associated with them.

Designing a Type Checker

When designing a Type Checker for a compiler here's the process:

- Identify the types that are available in the language.
- Identify the language construct that have types associated with them.
- Identify the semantic rules for the language.

Designing a Type Checker

When designing a Type Checker for a compiler here's the process:

- Identify the types that are available in the language.
- Identify the language construct that have types associated with them.
- Identify the semantic rules for the language.
- If a problem found, e.g. one tries to add a character to a double in C, we encounter a type error.

Designing a Type Checker

When designing a Type Checker for a compiler here's the process:

- Identify the types that are available in the language.
- Identify the language construct that have types associated with them.
- Identify the semantic rules for the language.
- If a problem found, e.g. one tries to add a character to a double in C, we encounter a type error.
- A language is considered ***strongly-typed*** if each and every type error is detected during compilation.
- Type checking can be done in compile time or in execution time.

Static Type Checking

- Static type checking is done at compile time. The information type checker needs is obtained via declarations and stored in a master symbol table.
- After this information is collected, the types involved in each operation are checked.

Static Type Checking

- Static type checking is done at compile time. The information type checker needs is obtained via declarations and stored in a master symbol table.
- After this information is collected, the types involved in each operation are checked.
- For example, if a and b are of type int and we assign very large values to them, $a * b$ may not be in the the acceptable range of ints, or an attempt to compute the ratio between two integers may raise a division by zero. These kind of type errors usually can not be detected at compile time.

Dynamic Type Checking

- Dynamic type checking is implemented by including type information for each data location at runtime.
- For example, a variable of type double would contain both the actual double value and some kind of tag indicating "double type".
- The execution of any operation begins by first checking these type tags. The operation is performed only if everything checks out. Otherwise, a type error occurs and usually halts execution.

Type Expressions

The type of a language construct will be denoted by a “type expression”.

The few basic type expressions are as follows:

- The basic types are boolean, char, integer, and real. A special basic type, type_error, will signal an error during type checking. Finally, a basic type void denoting “the absence of a value ” allows statements to be checked.
- Type expression may be named, a type name is a type expression.

Type Expressions

The type of a language construct will be denoted by a “type expression”.

The few basic type expressions are as follows:

- The basic types are boolean, char, integer, and real. A special basic type, type_error, will signal an error during type checking. Finally, a basic type void denoting “the absence of a value ” allows statements to be checked.
- Type expression may be named, a type name is a type expression.
- A type constructor applied to type expression is a type expression. Constructors include:
  - Arrays
  - Products
  - Records
  - Pointers
  - Functions

Arrays:

If T is a Type expression, then array(I, T) is a type expression denoting the type of an array with elements of type *T* and index set *I*. *I* is often a range of integers. For example

- var A: array[1..10] of integer;

Associates the type expression: **array(1..10, integer) with A**

Products

If T_1 and T_2 are type expressions, their Cartesian product $T_1 \times T_2$ is a type expression.

Records

The record type constructor will be applied to a tuple formed from field names and field types.

type	row =	record	
		address:	integer;
		lexeme:	array [1..15] of char
		end ;	
var	table:	array[1..10] of row;	

declares the name row representing the type expression **record((address** $\times$ **integer)** $\times$ **(lexeme** $\times$ **array(1 .. 15, char)))**

The variable *table* to be an array of records of this type.

Pointers:

If T is a type expression, then $pointer(T)$ is a type expression denoting the type “pointer to an object of type T”.

For example,

var p: $\uparrow$ row

declares variable p to have type pointer (row).

Functions:

Mathematically, a function maps elements of one set, the domain, to another set, the range. We may treat functions in programming languages as mapping a *domain type* D to a *range type* R. The type of such a function will be denoted by $D \rightarrow R$.

As for example,

function f(a, b : char): $\uparrow$ integer;

The type of f is denoted by the type expression

$$char \times char \rightarrow pointer(integer)$$

Specification of a simple Type Checker

The following grammar generate programs, represented by the nonterminal P, consisting of a sequence of declarations D followed by a single expression E.

$P \rightarrow D; E$

$D \rightarrow D; D | id : T$

$T \rightarrow char | integer | array[num] of \ T | \uparrow T$

$E \rightarrow literal \ | num | \ id | \ EmodE \ | E[E] \ | E \uparrow$

Table: Translation Scheme that saves the type of an identifier

Productions	Associated rules for type
$P \rightarrow D; E$	
$D \rightarrow D; D$	
$D \rightarrow id : T$	{ addtype(id.entry, T.type)}
$T \rightarrow char$	{ T.type = char }
$T \rightarrow integer$	{ T.type = integer }
$T \rightarrow \uparrow T_1$	{ T.type = pointer($T_{1.type}$) }
$T \rightarrow array[num]\ of\ T_1$	{ T.type = array($1 \ldots num.val$, $T_{1.type}$ }

Type Checking of Expressions

Table: Associated rules for Type Checking

Productions	Associated rules for type
$E \rightarrow literal$	E.type = char
$E \rightarrow num$	E.type = integer
$E \rightarrow id$	E.type = lookup(id.entry)
$E \rightarrow E_1 mod E_2$	E.type = **if** $E_1.type = integer$ **and** $E_2.type = integer$ **then** *integer* **else** *type_error*
$E \rightarrow E_1[E_2]$	E.type = **if** $E_{2.type} = integer$ **and** $E_{1.type} = array(s, t)$ **then** t **else** *type_error*
$E \rightarrow E_1 \uparrow$	E.type = if $E_{1.type}$=pointer(t) then t **else** *type_error*

Type Checking of Statements

The state statements we consider are assignment, conditional, and while statements.

Table: default

Productions	Associated rules for type
$S \rightarrow id = E$	{S.type = **if** id.type == E.type **then** void **else** *type_error* }
$S \rightarrow$ *if* E then S_1	{S.type = **if** E.type == Boolean **then** $S_1.type$ **else** *type_error* }
$S \rightarrow$ *while* E do S_1	{S.type = **if** E.type == Boolean **then** $S_1.type$ **else** *type_error* }
$S \rightarrow S_1 ; S_2$	{S.type = **if** S_1.type == void **and** S_2.type == void **then** void **else** *type_error* }

Type Checking of Functions

Productions	Associated actions
$T \rightarrow T_1 \; ' \rightarrow ' \; T_2$	$\{ T.type = T_1.type \rightarrow T_2.type \}$
$E \rightarrow E_1(E_2)$	{E.type = if $E_2.type$ == s and E_1.type == $s \rightarrow t$ **then** t **else** *type_error* }