

Context-sensitive analysis

What context-sensitive questions might the compiler ask?

1. Is \mathbf{x} a scalar, an array, or a function?
2. Is \mathbf{x} declared before it is used?
3. Are any names declared but not used?
4. Which declaration of \mathbf{x} does this reference?
5. Is an expression *type-consistent*?
6. Does the dimension of a reference match the declaration?
7. Where can \mathbf{x} be stored? (*heap, stack, ...*)
8. Does $\mathbf{*p}$ reference the result of a `malloc()`?
9. Is \mathbf{x} defined before it is used?
10. Is an array reference *in bounds*?
11. Does function `foo` produce a constant value?

These cannot be answered with a context-free grammar

Context-sensitive analysis

Why is context-sensitive analysis hard?

- need non-local information
- answers depend on values, not on syntax
- answers may involve computation

How can we answer these questions?

1. use context-sensitive grammars
 - general problem is P-space complete
2. use attribute grammars
 - augment context-free grammar with rules
 - calculate attributes for grammar symbols
3. use *ad hoc* techniques
 - augment grammar with arbitrary code
 - execute code at corresponding reduction
 - store information in attributes, symbol tables

Attribute grammars

Attribute grammar

- generalization of context-free grammar
- each grammar symbol has an associated set of attributes
- augment grammar with rules that define values
- high-level specification, independent of evaluation scheme

Dependencies between attributes

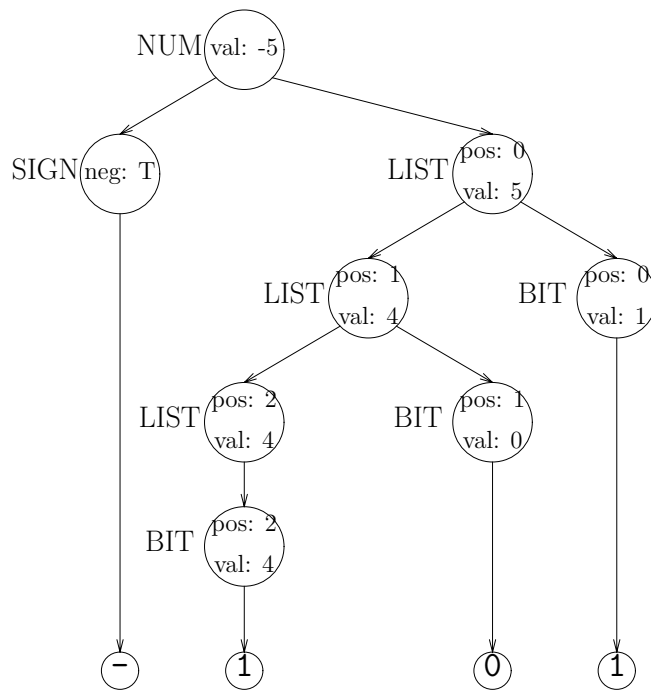
- values are computed from constants & other attributes
- *synthesized attribute* – value computed from children
- *inherited attribute* – value computed from siblings & parent

Example attribute grammar

A grammar to evaluate signed binary numbers *due to Scott K. Warren, Rice Ph.D.*

Production	Evaluation Rules
1 NUM ::= SIGN LIST	LIST.pos \leftarrow 0 NUM.val \leftarrow if SIGN.neg then -LIST.val else LIST.val
2 SIGN ::= +	SIGN.neg \leftarrow false
3 SIGN ::= -	SIGN.neg \leftarrow true
4 LIST ::= BIT	BIT.pos \leftarrow LIST.pos LIST.val \leftarrow BIT.val
5 LIST ₀ ::= LIST ₁ BIT	LIST ₁ .pos \leftarrow LIST ₀ .pos + 1 BIT.pos \leftarrow LIST ₀ .pos LIST ₀ .val \leftarrow LIST ₁ .val + BIT.val
6 BIT ::= 0	BIT.val \leftarrow 0
7 BIT ::= 1	BIT.val \leftarrow 2 ^{BIT.pos}

Example attribute grammar



- `val` and `neg` are *synthesized* attributes
- `pos` is an *inherited* attribute

Syntax-directed translation

Disadvantages of attribute grammars

- handling non-local information, locating answers
- storage management, avoiding circular evaluation

Syntax-directed translation

- allow arbitrary actions
- provide central repository
- can place actions amid production

Examples

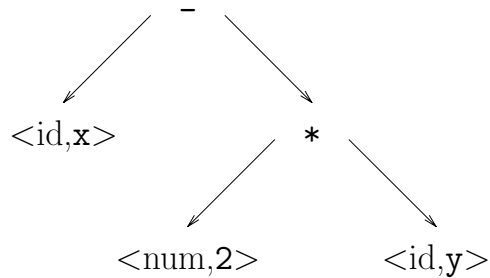
- YACC — $A ::= B C \{ \$\$ = \text{concat}(\$1, \$2); \}$
- CUP — $A:n ::= B:m C:p \{ : n = \text{concat}(m, p); : \}$

Typical uses

- build abstract syntax tree & symbol table
- perform error/type checking

Abstract syntax tree

An abstract syntax tree (AST) is the procedure's parse tree with the nodes for most non-terminal symbols removed.



This represents “**x** - 2 * **y**”.

For ease of manipulation, can use a linearized (operator) form of the tree.

x **2** **y** * - in postfix form.

A popular *intermediate representation*.

Symbol tables

A *symbol table* associates values or attributes (e.g., *types and values*) with names.

What should be in a symbol table?

- variable and procedure names
- literal constants and strings

What information might compiler need?

- textual name
- data type
- declaring procedure
- lexical level of declaration
- if array, number and size of dimensions
- if procedure, number and type of parameters

Symbol tables

Implementation

- usually implemented as hash tables

How to handle nested lexical scoping?

- when we ask about a name, we want the closest lexical declaration

One solution

- use one symbol table per scope
- tables chained to enclosing scopes
- insert names in table for current scope
- name lookup starts in current table if needed, checks enclosing scopes in order

Type systems

Types

- values that share a set of common properties
- defined by language and/or programmer

Type system

1. set of types in a programming language, and
2. rules that use types to specify program behavior

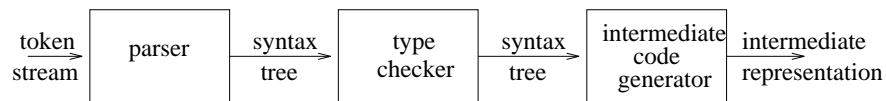
Example type rules

- If operands of addition are of type integer, then result is of type integer
- The result of the unary & operator is a pointer to the object referred to by the operand

Advantages of typed languages

- ensure run-time safety
- expressiveness (overloading, polymorphism)
- provide information for code generation

Type checking



Type checker

- enforces rules of type system
- may be strong/weak, static/dynamic

Static type checking

- performed at compile time
- early detection, no run-time overhead
- not always possible (e.g., $A[i]$)

Dynamic type checking

- performed at run time
- more flexible, rapid prototyping
- overhead to check run-time type tags

Type expressions

Type expressions

- used to represent the type of a language construct
- describes both language and programmer types

Examples

- basic types: integer, real, character, ...
- constructed types: arrays, records, pointers, functions,...

Constructing new types

- arrays
- records
- pointers
- functions

$array(1..10, T)$

$T_1 \times T_2 \times \dots$

$pointer(T)$

$T_1 \times T_2 \times \dots \rightarrow T_n$

A simple type checker

Using a synthesized attribute grammar, we will describe a type checker for arrays, pointers, statements, and functions.

Grammar for source language:

$P ::= D ; E$
 $D ::= D ; E \mid \text{id} : T$
 $T ::= \text{char} \mid \text{integer} \mid \text{array} [\text{num}] \text{ of } T \mid \uparrow T$
 $E ::= \text{literal} \mid \text{num} \mid \text{id} \mid E \text{ mod } E \mid E[E] \mid E \uparrow$

- Basic types *char*, *integer*, *typeError*
- assume all arrays start at 1, e.g.,
 array [256] of char
 results in the type expression *array(1..256,char)*
- \uparrow builds a pointer type, so \uparrow integer
 results in the type expression *pointer(integer)*

Type checking example

Partial attribute grammar for the type system

$D ::= \text{id} : T \quad \{ \text{addtype}(\text{id.entry}, T.type) \}$
 $T ::= \text{char} \quad \{ T.type \leftarrow \text{char} \}$
 $T ::= \text{integer} \quad \{ T.type \leftarrow \text{integer} \}$
 $T ::= \uparrow T_1 \quad \{ T.type \leftarrow \text{pointer}(T_1.type) \}$
 $T ::= \text{array} [\text{num}] \text{ of } T_1 \quad \{ T.type \leftarrow \text{array}(1 \dots \text{num.val}, T_1.type) \}$

Type checking expressions

Each expression is assigned a type using rules associated with the grammar.

$E ::= \text{literal}$	$\{ E.type \leftarrow char \}$
$E ::= \text{num}$	$\{ E.type \leftarrow integer \}$
$E ::= \text{id}$	$\{ E.type \leftarrow lookup(id.entry) \}$
$E ::= E_1 \text{ mod } E_2$	$\{ E.type \leftarrow \text{if } E_1.type = integer \text{ and } E_2.type = integer \text{ then } integer \text{ else } typeError \}$
$E ::= E_1[E_2]$	$\{ E.type \leftarrow \text{if } E_2.type = integer \text{ and } E_1.type = array(s,t) \text{ then } t \text{ else } typeError \}$
$E ::= E_1 \uparrow$	$\{ E.type \leftarrow \text{if } E_1.type = pointer \text{ then } t \text{ else } typeError \}$

Type checking statements

Statements do not typically have values, therefore we assign them the type *void*. If an error is detected within the statement, it gets type *typeError*.

$S ::= \text{id} \leftarrow E$	$\{ S.type \leftarrow \text{if } id.type = E.type \text{ then } void \text{ else } typeError \}$
$S ::= \text{if } E \text{ then } S_1$	$\{ S.type \leftarrow \text{if } E.type = boolean \text{ then } S_1.type \text{ else } typeError \}$
$S ::= \text{while } E \text{ do } S_1$	$\{ S.type \leftarrow \text{if } E.type = boolean \text{ then } S_1.type \text{ else } typeError \}$
$S ::= S_1 ; S_2$	$\{ S.type \leftarrow \text{if } S_1.type = void \text{ then } void \text{ else } typeError \}$

Type checking functions

We add two new productions to the grammar to represent function declarations and applications

$$\begin{array}{ll} T ::= T \rightarrow T & \text{declaration} \\ E ::= E (E) & \text{application} \end{array}$$

To capture the argument and return type, we use

$$\begin{array}{l} T ::= T_1 \rightarrow T_2 \{ T.type \leftarrow (T_1.type \rightarrow T_2.type) \} \\ E ::= E_1 (E_2) \{ E.type \leftarrow \text{if } E_1.type = s \rightarrow t \\ \quad \text{and } E_2.type = s \text{ then } t \\ \quad \text{else } typeError \} \end{array}$$