

## Context-sensitive analysis

---

What context-sensitive questions might the compiler ask?

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

## 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

Dependences between attributes

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

## 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

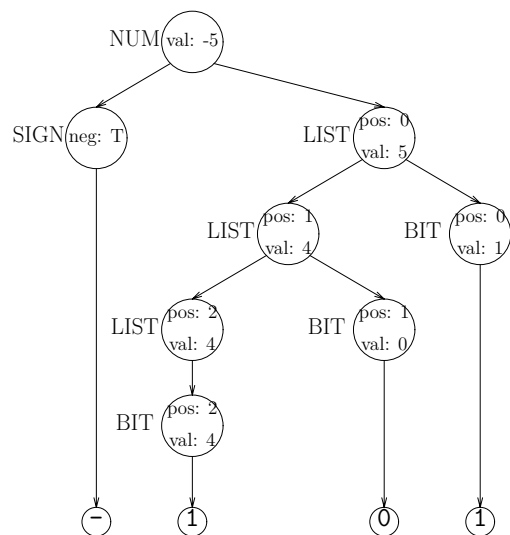
## Example attribute grammar

---

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 $\leftarrow$ <b>false</b>
3 SIGN ::= -	SIGN.neg $\leftarrow$ <b>true</b>
4 LIST ::= BIT	BIT.pos $\leftarrow$ LIST.pos LIST.val $\leftarrow$ BIT.val
5 LIST <sub>0</sub> ::= LIST <sub>1</sub> BIT	LIST <sub>1</sub> .pos $\leftarrow$ LIST <sub>0</sub> .pos + 1 BIT.pos $\leftarrow$ LIST <sub>0</sub> .pos LIST <sub>0</sub> .val $\leftarrow$ LIST <sub>1</sub> .val + BIT.val
6 BIT ::= 0	BIT.val $\leftarrow$ 0
7 BIT ::= 1	BIT.val $\leftarrow$ 2 <sup>BIT.pos</sup>

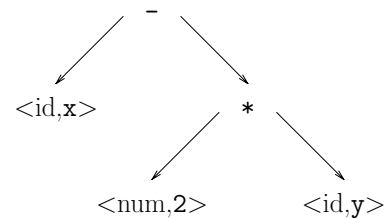
## Example attribute grammar



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

## 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*.

## 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

## 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?

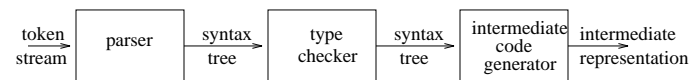
- when we ask about a name, we want the closest lexical declaration

### 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 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 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 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  $array(1..10, T)$
- records  $T_1 \times T_2 \times \dots$
- pointers  $pointer(T)$
- functions  $T_1 \times T_2 \times \dots \rightarrow T_n$

## 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 | id: T
T ::= char | integer | array [num] of T | ↑ T
E ::= literal | num | id | E mod E | E[E] | E ↑
```

- 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)*
- ↑ builds a pointer type, so ↑ integer  
results in the type expression *pointer(integer)*

## Type checking expressions

---

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

```
E ::= literal      { E.type ← char }
E ::= num         { E.type ← integer }
E ::= id         { E.type ← lookup(id.entry) }
E ::= E1 mod E2 { E.type ← if E1.type = integer and
                        E2.type = integer then integer
                        else typeError }
E ::= E1[E2]     { E.type ← if E2.type = integer and
                        E1.type = array(s,t) then t
                        else typeError }
E ::= E1 ↑       { E.type ← if E1.type = pointer
                        then t else typeError }
```

## Type checking example

---

Partial attribute grammar for the type system

```
D ::= id: T      { addtype(id.entry, T.type) }
T ::= char      { T.type ← char }
T ::= integer   { T.type ← integer }
T ::= ↑ T1     { T.type ← pointer(T1.type) }
T ::= array [num] of T1 { T.type ←
                        array(1..num.val, T1.type) }
```

## 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 ::= id ← E     { S.type ← if id.type = E.type
                    then void
                    else typeError }
S ::= if E then S1 { S.type ← if E.type = boolean
                    then S1.type
                    else typeError }
S ::= while E do S1 { S.type ← if E.type = boolean
                    then S1.type
                    else typeError }
S ::= S1 ; S2   { S.type ← if S1.type = void
                    then void
                    else typeError }
```

