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

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

**Dependences 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.*

<table>
<thead>
<tr>
<th>Production</th>
<th>Evaluation Rules</th>
</tr>
</thead>
</table>
| 1 NUM ::= SIGN LIST | LIST.pos ← 0  
NUM.val ← if SIGN.neg  
then -LIST.val  
else LIST.val |
| 2 SIGN ::= + | SIGN.neg ← false |
| 3 SIGN ::= - | SIGN.neg ← true |
| 4 LIST ::= BIT | BIT.pos ← LIST.pos  
LIST.val ← BIT.val |
| 5 LIST₀ ::= LIST₁ BIT | LIST₁.pos ← LIST₀.pos + 1  
BIT.pos ← LIST₀.pos  
LIST₀.val ← LIST₁.val + BIT.val |
| 6 BIT ::= 0 | BIT.val ← 0 |
| 7 BIT ::= 1 | BIT.val ← 2BIT.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 { $$ = concat($1,$2); }`
- CUP — `A:n ::= B:m C:p { n = 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.

```
- <id,x> * <num,2> <id,y>
```

This represents “\(x - 2 \times y\)”.

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
  \[ \text{array}(1 \ldots 10, \ T) \]
- records
  \[ T_1 \times T_2 \times \ldots \]
- pointers
  \[ \text{pointer}(T) \]
- functions
  \[ T_1 \times T_2 \times \ldots \rightarrow T_n \]
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 example

Partial attribute grammar for the type system

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

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

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

Type checking statements

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

\[
\begin{align*}
S &::= \text{id} \leftarrow E \quad \{ \ S.\text{type} \leftarrow \text{if } \text{id.type} = E.\text{type} \\
& \quad \text{then void} \\
& \quad \text{else typeError} \} \\
S &::= \text{if } E \text{ then } S_1 \quad \{ \ S.\text{type} \leftarrow \text{if } E.\text{type} = \text{boolean} \\
& \quad \text{then } S_1.\text{type} \\
& \quad \text{else typeError} \} \\
S &::= \text{while } E \text{ do } S_1 \quad \{ \ S.\text{type} \leftarrow \text{if } E.\text{type} = \text{boolean} \\
& \quad \text{then } S_1.\text{type} \\
& \quad \text{else typeError} \} \\
S &::= S_1 ; S_2 \quad \{ \ S.\text{type} \leftarrow \text{if } S_1.\text{type} = \text{void} \\
& \quad \text{then void} \\
& \quad \text{else typeError} \} \\
\end{align*}
\]
We add two new productions to the grammar to represent function declarations and applications

\[
\begin{align*}
T & ::= T \rightarrow T \quad \text{declaration} \\
E & ::= E \left( E \right) \quad \text{application}
\end{align*}
\]

To capture the argument and return type, we use

\[
\begin{align*}
T & ::= T_1 \rightarrow T_2 \quad \{ T.type \leftarrow (T_1.type \rightarrow T_2.type) \} \\
E & ::= E_1 \left( E_2 \right) \quad \{ E.type \leftarrow \text{if } E_1.type = s \rightarrow t \\
& \quad \text{and } E_2.type = s \text{ then } t \\
& \quad \text{else } typeError \}
\end{align*}
\]