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 \texttt{malloc()}?
9. Is \( x \) defined before it is used?
10. Is an array reference in bounds?
11. Does function \texttt{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
- \textit{synthesized attribute} – value computed from children
- \textit{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.

\[
\begin{array}{c|c|c}
\text{Production} & \text{Evaluation Rules} \\
\hline
1 \text{ NUM ::= SIGN LIST} & \text{LIST.pos} \leftarrow 0 \\
& \text{NUM.val} \leftarrow \text{if SIGN.neg then } -\text{LIST.val} \\
& \text{else LIST.val} \\
2 \text{ SIGN ::= +} & \text{SIGN.neg} \leftarrow \text{false} \\
3 \text{ SIGN ::= -} & \text{SIGN.neg} \leftarrow \text{true} \\
4 \text{ LIST ::= BIT} & \text{BIT.pos} \leftarrow \text{LIST.pos} \\
& \text{LIST.val} \leftarrow \text{BIT.val} \\
5 \text{ LIST_0 ::= LIST_1 BIT} & \text{LIST_1.pos} \leftarrow \text{LIST_0.pos} + 1 \\
& \text{BIT.pos} \leftarrow \text{LIST_0.pos} \\
& \text{LIST_0.val} \leftarrow \text{LIST_1.val} + \text{BIT.val} \\
6 \text{ BIT ::= 0} & \text{BIT.val} \leftarrow 0 \\
7 \text{ BIT ::= 1} & \text{BIT.val} \leftarrow 2 \times \text{BIT.pos}
\end{array}
\]
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.

```
  - 
     |     |     |
     <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*.

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 — \(A:n ::= B:m C:p \{ : n = concat(m,p); : \}\)

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 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
  \(array(1 \ldots 10,\ T)\)
- records
  \(T_1 \times T_2 \times \ldots\)
- pointers
  \(pointer(T)\)
- 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:

\[
P ::= D ; E \\
D ::= D ; E \mid id \colon T \\
T ::= char \mid integer \mid array [num] \mid T \\
E ::= literal \mid num \mid id \mid E mod E \mid E[E] \mid E \\
\]

- Basic types char, integer, typeError
- assume all arrays start at 1, e.g.,
  array [256] of char
  results in the type expression \textit{array}(1\ldots256,char)
- \textit{\uparrow} builds a pointer type, so \textit{\uparrow} integer
  results in the type expression \textit{pointer}(integer)

Type checking expressions

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

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

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}.

\[
S ::= \text{id }\leftarrow E \quad \{ S.type \leftarrow \text{if } \text{id.type }= E.type \\
\text{then } \text{void} \\
\text{else } \text{typeError} \} \\
S ::= \text{if } E \text{ then } S_1 \quad \{ S.type \leftarrow \text{if } E.type = \text{boolean} \\
\text{then } S_1.type \\
\text{else } \text{typeError} \} \\
S ::= \text{while } E \text{ do } S_1 \quad \{ S.type \leftarrow \text{if } E.type = \text{boolean} \\
\text{then } S_1.type \\
\text{else } \text{typeError} \} \\
S ::= S_1 ; S_2 \quad \{ S.type \leftarrow \text{if } S_1.type = \text{void} \\
\text{then } \text{void} \\
\text{else } \text{typeError} \} \\
\]
Type checking functions

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

\[ T ::= T \rightarrow T \quad \text{declaration} \]
\[ E ::= E ( E ) \quad \text{application} \]

To capture the argument and return type, we use

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