Context-sensitive analysis

1. What context-sensitive questions might the compiler ask?
   1. Is \( x \) a scalar, an array, or a function?
   2. Where can \( x \) be stored? (heap, stack, ...?)
   3. Where can \( x \) be stored? (heap, stack, ...?)
   4. Does the declaration of \( x \) produce a constant value?
   5. Does the declaration of \( x \) produce a constant value?
   6. Does the declaration of \( x \) produce a constant value?
   7. Does the declaration of \( x \) produce a constant value?
   8. Does the declaration of \( x \) produce a constant value?
   9. Does the declaration of \( x \) produce a constant value?
   10. Is an array reference in bounds?

Context-sensitive analysis

Why is context-sensitive analysis hard?

Store information in attributes, symbolic tables.

- Execute code at corresponding reduction.
- Maintain grammar with arbitrary code.
- Use ad hoc techniques.

Calculate attributes for grammar symbols.

- Maintain context-free grammar with rules.

2. Use attribute grammars.

General problem is P-space complete.

1. Use context-sensitive grammars.

How can we answer these questions?

- Answers may involve computation.
- Answers depend on values, not on syntax.
- Need non-local information.

Why is context-sensitive analysis hard?
Attributed grammars

Generalization of context-free grammar

Each grammar symbol has an associated set of attributes

Evaluation scheme

High-level specification, independent of attributes

Evaluation of context-free grammar

A grammar to evaluate signed binary numbers

Example attribute grammar
Attributed grammars

Examples

YACC

CUP

Typical uses

- perform error/type checking
- build symbol table
- handle non-local information

Typical disadvantages of attribute grammars

- Syntax-directed translation
- avoid circular evaluation
- store answers
- handle non-local information

Disadvantages of attribute grammars

- hand and neg are synthesized attributes
- pos is an inherited attribute

- Syntax/directed translation
- allow arbitrary actions
- provide central repository
- avoid circular evaluation

Lecture 9, Page 5
**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
As a simple type checker,

A simple type checker

functions

functions

pointers

pointers

records

records

arrays

arrays

Constructing new types

Constructing new types

... Examples

Examples

Type expressions

Type expressions

Type expressions

Basic types

Basic types

Assume all arrays start at 1, e.g.,

Assume all arrays start at 1, e.g.,

Basic types char, integer, integer

Basic types char, integer, integer

Grammar for source language:

Grammar for source language:

grammar for source language

Grammar for source language

Statements and functions,

Statements and functions,

describes both language and programmer types,

describes both language and programmer types,

Using a synthesized attribute grammar, we will

Using a synthesized attribute grammar, we will
Type checking expressions

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

Partial attribute grammar for the type system
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 = if id /= E then void else typeError

S := while E do S

S := if !E then S := void

S := if id /= E then void
```

Type checking functions

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

```
T := T
E := E
```

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