LANGUAGE PROCESSORS

UNIT 7: SEMANTIC ANALYSIS

David Griol Barres
dgriol@inf.uc3m.es
Computer Science Department
Carlos III University of Madrid
Leganés (Spain)

OUTLINE

- Introduction
- Semantic Analysis
- Type Checking
- Designing a Type Checker
- Example of a simple language
- Cycles in Representations of types
- Scope Checking

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Introduction

- "He am a driver" might be syntactically correct but semantically wrong.
- Semantic analysis in the compiler's last chance to detect incorrect programs.

tokens → Parse tree → Annotated parse tree

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Semantic Analysis

- For a program to be semantically valid:
  - All variables, functions and classes properly defined.
  - Expressions and variables must be used in ways that respect the type system.
  - Access control must be respected.
  - …

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Semantic Analysis

- Semantic information is associated to the programming language constructs by attaching attributes to the grammar.
- Values for attributes are computed by semantic rules associated with grammar constructors.
- This phase modifies the symbol table and is often mixed with the intermediate code generation.
- The method applied for the determination of syntactic and semantic correctness of a given program is called **static checking**.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Semantic Analysis

- Notations for associating semantic rules with productions:
  - Syntax-directed definitions.
  - Translation schemes.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Semantic Analysis

- Examples
  - **Type checking**
    - errors messages are issued if an operator is applied to an incompatible operand.
  - **Flow-of-control checks**
    - for example, in C a break statement occurs inside a while, or for statements.
  - **Uniqueness checks**
    - sometimes, an object must be defined exactly one, for example, labels in a case statement must be distinct.
  - **Name-related checks**
    - sometimes, in a construct the same name must appear two or more times.

Type Checking

- **Objective:** Verify that each operation executed in a program respect the type system of the program.

- **Static type checking:**
  - Done at compile-time.
  - Information obtained from a master symbol table.

- **Dynamic type checking:**
  - Type information is included for each data location at runtime.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Designing a Type Checker

- Identify the types in the input language.
- Identify the language constructs that have types associated to them.
- Identify the semantic rules for the language.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Example of a simple language

- Simple language has a static type system
  - Goal: check the types of all the expressions in the language
- Types in the language:
  - Basic types
    - char
    - integer
    - type_error
  - Complex types
    - Array[n] of T (T, 1..n)
    - ^T pointer

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

A simple language

- Grammar:

P → D ; E
D → D ; D | id : T
T → char | integer | array [num] of T | ^T
E → literal | num | id | E mod E | E [E] | E^

  - D for declaration
  - T for type
  - E for expression

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

A simple language

- Semantic actions:

P → D ; E	
D → D ; D	
D → id : T	{ addType (id.entry, T.type) }
T → char	{ T.type := char }
T → integer	{ T.type := integer }
T → ^T_1	{ T_0.type := pointer (T_1.type) }
T → array [num] of T	{ T_0.type := array(1..num.val, T_1.type) }

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

A simple language

- Type checking of expressions:
  - Constants

E → literal	{ E.type := char }
E → num	{ E.type := integer }

  - Identifiers

E → id	{ E.type := lookup (id.entry) }

  - Operators

E → E mod E	{ if (E_1.type = integer) and (E_2.type = integer) E_0.type = integer else E_0.type = type_error }

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

A simple language

- Arrays:

$E \rightarrow E\,[\,E\,]$ { if (E_1.type = integer) and
(E_2.type = array(s, t)) then
E_0.type = t
else E_0.type = type_error }

- Pointers:

$E \rightarrow E\,\hat{}$ { if (E_1.type = pointer(t))
E_0.type = t
else E_0.type = type_error }

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Type checking of statements

- By default, a sentence is of void type unless it is an error
  - if and while:

$S \rightarrow id := E$ { if (id.type = E.type) then S_0.type = void
else S_0.type := type_error }

$S \rightarrow$ if E then S { if (E.type = boolean) then S.type = S_1.type
else S_0.type := type_error }

$S \rightarrow$ while E do S { if (E.type = boolean) then S.type = S_1.type
else S_0.type := type_error }

$S \rightarrow S \; ; S$ { if (S_1.type = void) and (S_2.type = void)
then S_0.type = void
else S_0.type := type_error }

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Type checking of functions

$T \rightarrow T$ '$\rightarrow$' T	$\{ T_0.type = T_1.type \rightarrow T_2.type \}$

Ej.: int f(double x, char y) { ... }

type de f: double x char $\rightarrow$ int

- Rule for checking the type of a function application

$E \rightarrow E (E)$	$\{$ if $(E_2.type = s)$ and $(E_1.type = s \rightarrow t)$ then $E_0.type = t$ else $E_0.type :=$ type_error $\}$

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Type Conversions

- Many compilers correct the simplest of type errors
  - x + y ?
  - What if x is of type real and y is of type integer?

$E \rightarrow E \text{ op } E$ { if (E_1.type = integer) and
(E_2.type = integer) then E_0.type = integer
else if (E_1.type = integer) and
(E_2.type = real) then E_0.type = real
else if (E_1.type = real) and
(E_2.type = integer) then E_0.type = real
else if (E_1.type = real) and
(E_2.type = real) then E_0.type = real
else E_0.type = type_error}

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Overloading of functions and operators

- An overloaded symbol has different meanings depending on its context
  - (4 + "a") and (4+'a') in Java
  - In Ada
    - function "*" (k,j: integer) return complex
    - function "*" (x,y: complex) return complex
  - Therefore

    3*5 either has type integer or complex, depending on its context

    2*(3*5) → integer

    z*(3*5) → complex

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Possible types of expressions

E' → E	{ E'.types=E.types}
E → id	{ E.types=lookup(id.entry)}
E → E (E)	{E_0.types= {t | there exists an s in E_2.types such that s→ t is in E_1.types}

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Cycles in Representations of types

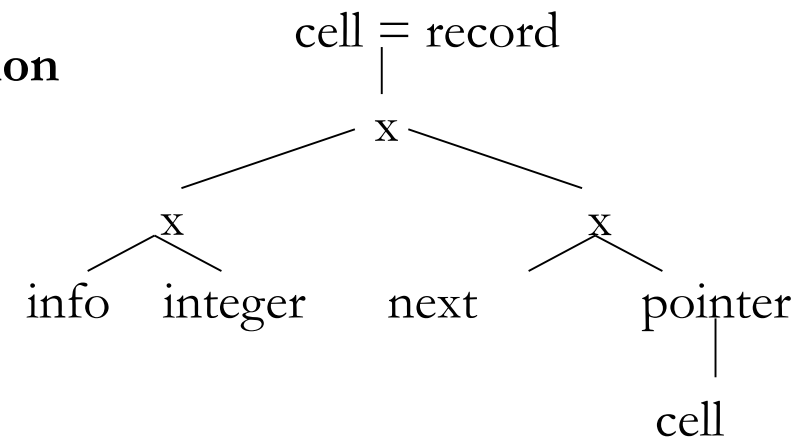
- Often data structures are defined recursively. Example.: lists

```
Type link = ^cell;
      cell = record
                  info : integer;
                  next: link
                end;
```

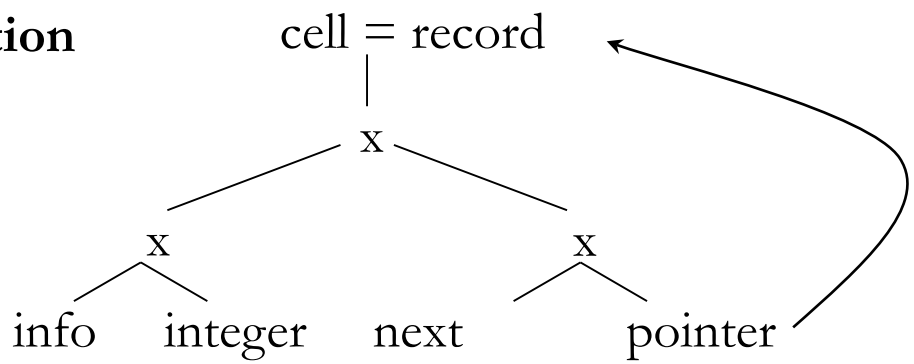

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Cycles in Representations of types

Acyclic representation



Cyclic representation



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Scope Checking

- The visibility of an identifier is constrained to some section of the program:
  - Global and local variables.
  - **Objective**: Determine if the identifier is accessible at a specific point in the program.
  - **Approaches:**
    - Use an individual symbol table for each scope (stack containing all the scopes).
    - A single global table for all the scopes (scope-number, var-name).