

Programming Language Implementation VIII

Semantic Analysis

Determines those non-syntactic properties that can be determined from the program text. It:

- typically determines the **kind** of each identifier
- performs **type checking** and **type inference**, and
- adds this information to the symbol table.

It also checks that

- variables are declared before use,
- variables are declared only once (within a particular scope),
- type compatibility and required coercions,
- matching of actual with formal parameters,
- resolves overloaded operator symbols
- ...

The reason we need a separate phase for semantic analysis is that syntactic analysis based on context-free grammars is not powerful enough to check these properties, since they are inherently **context sensitive**.

In this and the next lecture we will look at **semantic analysis** and **program optimisation** and in particular at

- **Attribute grammars**
- **Type inference**
- **Common program optimisations**

The material is (loosely) based on Aho et al Chapter 5.

Attribute Grammars

Semantic analysis is often done in an ad hoc manner, but **attribute grammars** can be used to formalize it.

Attribute grammars are due to Knuth in 1968. They extend normal context-free grammars by adding attributes to non-terminals. These attributes are mainly used for two purposes:

- to compute structures (e.g. syntax-trees) to be returned by the grammar and
- to steer the application of productions by providing additional information when calling a production.

We can think of each node of the parse tree X as a record with attributes acting as fields. We use $X.a$ to refer to the value of attribute a .

Attributes are divided into **inherited** attributes and **synthesized** attributes. Synthesized attributes are computed by a production. Inherited attributes are provided when a production is called and tested or used to compute synthesized attributes.

Attribute Grammars (Cont.)

An attribute grammar consists of a **context-free grammar** plus **rules** for assigning values to attributes of symbols in the parse tree.

With each grammar production $A \rightarrow X_1 \dots X_n$ we associate a set of semantic rules of the form

$$b := f(c_1, \dots, c_n)$$

where f is a (usually side effect-free) function, and b and c_1, \dots, c_n are attributes of the symbols in the rule. Such a rule is said to define b .

We require rules to define each synthesized attribute of A and to define all inherited attributes of the symbols $X_1 \dots X_n$.

More generally, a production can also specify

- conditions on attribute values for a production to be applicable,
- procedures to be evaluated which, for example, might update the symbol table.

A parse tree showing the values of the attributes at each node is said to be annotated or decorated.

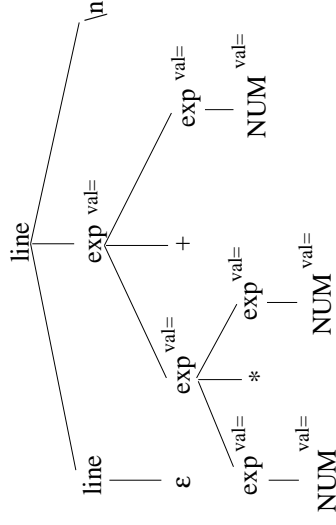
Attribute Grammar – Simple Example

We have already seen an example of an attribute-grammar:

Production	Semantic Rules
$line \rightarrow \epsilon$	
$line \rightarrow line\ exp\ \backslash n$	$print(exp.val)$
$exp_0 \rightarrow exp_1 + exp_2$	$exp_0.val := exp_1.val + exp_2.val$
$exp_0 \rightarrow exp_1 - exp_2$	$exp_0.val := exp_1.val - exp_2.val$
$exp_0 \rightarrow exp_1 * exp_2$	$exp_0.val := exp_1.val * exp_2.val$
$exp_0 \rightarrow exp_1 / exp_2$	$exp_0.val := exp_1.val / exp_2.val$
$exp_0 \rightarrow (exp_1)$	$exp_0.val := exp_1.val$
$exp_0 \rightarrow NUM$	$exp_0.val := NUM.val$

In this grammar exp and NUM have a single synthesized attribute val while $line$ and the remaining terminal symbols have no attributes.

The annotated parse tree for $3 * 5 + 4 \backslash n$ is:



More Complex Attribute Grammar Example

Consider a simple desktop calculator language which has integer and real variables, assignment and arithmetic expressions.

$program$	\rightarrow	$declarations\ statements$
$declarations$	\rightarrow	ϵ
$declaration$	\rightarrow	$declaration ; declarations$
		$int\ IDENT$
		$real\ IDENT$
$statements$	\rightarrow	ϵ
		$assign ; statements$
$assign$	\rightarrow	$IDENT := exp$
exp	\rightarrow	$exp + exp$
		$exp - exp$
		exp / exp
		$exp * exp$
		$exp (exp)$
		$\sim exp$
		$IDENT$
		$REALCONST$
		$INTCONST$

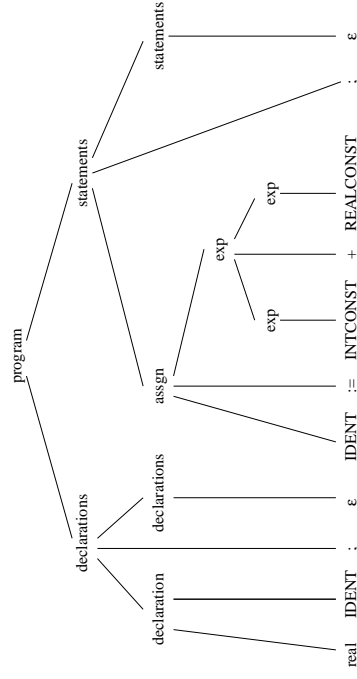
The terminal symbols are $;$, $real$, int , $+$, $-$, $/$, $*$, $($, $)$, \sim , $IDENT$, $REALCONST$, $INTCONST$ while the remaining symbols are non-terminals and $program$ is the start symbol.

Attribute Grammar Example (Cont.)

An example program is

```
real x;
x := 100 + 8.9;
```

This has the parse tree



Attribute Grammar Example (Cont.)

The following attribute grammar performs type checking and resolves overloading of arithmetic operations and determines where coercion is required. Not all rules are given.

Every symbol has an environment attribute *env* which is an array containing type information about identifiers. It is a synthesized attribute for *declaration* and *declarations* and inherited for *statements*, *assign* and *exp*.

- The index into *env* is the unique *id* attribute of an identifier created during screening.
- The function *lookup(env, IDENT.id)* returns the type of the identifier. If the identifier is not in *env* it returns *null*.
- The function *update(env, IDENT.id, type)* returns a new environment which is the same as *env* except that the type of the identifier is set to *type*.

declaration and *declarations* have an additional inherited attribute *envin* which is the input environment.

Expressions have three additional attributes:

- *opn* – the arithmetic operation to be performed (synthesized)
- *type* – the actual type of the expression (synthesized)
- *reqtype* – the type required by the surrounding expression (inherited)

Coercion is needed if *type* ≠ *reqtype*.

Integers can be coerced to reals but not vice versa.

Attribute Grammar Example (Cont.)

The following attribution rules records the type (real or int) of an identifier and checks that no identifier is defined more than once

Grammar rule: $declarations1 \rightarrow declaration ; declarations2$
Attribution Rules:

```
declaration.envin := declarations1.envin
declarations2.envin := declaration.env
declarations1.env := declarations2.env
```

Grammar rule: $declarations \rightarrow \epsilon$
Attribution Rules:

```
declarations.env := declarations.envin
```

Grammar rule: $declaration \rightarrow int IDENT$

Attribution Rules:

```
declaration.env :=
  update(declaration.envin,IDENT.id,int)
```

Condition:

```
lookup(env,IDENT.id) = null
```

Attribute Grammar Example (Cont.)

The following attribution rules type check the arithmetic operations in an assignment statements, resolve overloading and determine if coercion is needed:

Grammar rule: $assign \rightarrow IDENT := exp$
Attribution Rules:

```
exp.env := assign.env
exp.reqtype := lookup(assign.env,IDENT.id)
assign.opn :=
  if exp.reqtype = int then int_asg else real_asg
```

Condition:

```
coercible(exp.type, exp.reqtype)
```

where $coercible(T_1, T_2)$ holds if $T_1 = T_2$, or $T_1 = int$ and $T_2 = real$.

Grammar rule: $exp1 \rightarrow exp2 + exp3$

Attribution Rules:

```
exp2.env := exp1.env
exp3.env := exp1.env
exp1.type :=
  if exp2.type = int and exp3.type = int
  then int else real
exp1.opn :=
  if exp1.type = int then int_add else real_add
exp2.reqtype := exp1.type
exp3.reqtype := exp1.type
```

Attribute Grammar Example (Cont.)

Grammar rule: $exp \rightarrow IDENT$

Attribution Rules:

```
exp.type := lookup(exp.env, IDENT.id)
exp.opn := nullOp
```

Exercise: Give the rules and conditions for the grammar rule:

$exp \rightarrow REALCONST$

Environment Attribute

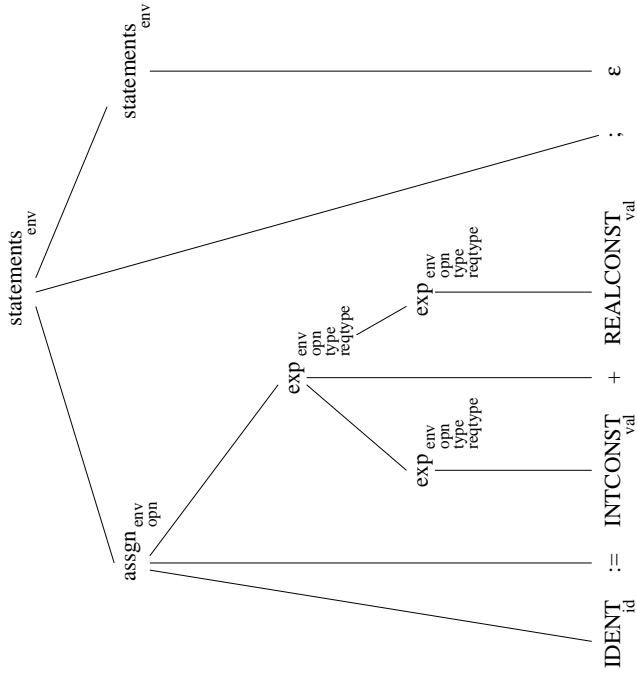
Conceptually the environment is an inherited attribute. In practice however, it is usually implemented as a single global variable, the symbol table. In a language with nested blocks it will be implemented using a stack.

Typical fields are:

- For a variable:
 - type and precision
 - address
 - if an array its dimension, subscript and component types
 - if a record its field names and types
 - if a parameter, how it is passed
- For a type:
 - its description
- For a procedure:
 - whether forward
 - parameter names and types
 - if function, the result type.

Attribute Dependency Graph

A sentence such as $x := 100 + 8.9$; has a **dependency graph** detailing how each attribute depends on the other attributes.



Computing Attributes

An attribute grammar is **well-defined** if every attribute is defined and for no sentence does the dependency graph contain a cycle. (ie it must be a **dag**—directed acyclic graph).

For every dag, we can give an ordered list of the attributes so that each attribute comes after those attributes on which it depends.

There are three main approaches to determining the order in which to compute the attributes. We will now look at these.

Computation of Attribute Evaluation Order – Method 1

We can compute the order of attribute computation dynamically from the particular parse-tree:

- Construct the dependency graph for the parse tree,
- Determine an order for evaluating the attributes so that an attribute's value is only determined after the value of the attributes it depends on have been determined. (Using **topological sorting** for instance).
- Evaluate the rules in this order

[(Ullman et al) calls this the parse tree method]

Computation of Attribute Evaluation Order – Method 2

Or we can compute the order statically by analysing the attribute grammar and then generating generic instructions for each production on how to compute attributes while traversing the parse tree. (Ullman et al) calls this the rule-based method.

This tree traversal can be specified by giving visit sequences for each rule. This details how to traverse the tree and which actions to take at each node during the traversal. The possible steps are:

- visit a child
- visit the parent
- evaluate an attribute
- evaluate a condition.

This is not as general as dynamic evaluation of the attribute computation order, but is less expensive at run-time and also checks that an attribute grammar is well-defined for all sentences.

Computing Attributes – Example

Grammar rule: $exp1 \rightarrow exp2 + exp3$

Attribution Rules:

```
exp2.env := exp1.env
exp3.env := exp1.env
exp1.type :=
  if exp2.type = int and exp3.type = int
  then int else real
exp1.opn :=
  if exp1.type = int then int_add else real_add
exp2.reqtype := exp1.type
exp3.reqtype := exp1.type
```

we might use the visit sequence

```
evaluate exp2.env
move to exp2
evaluate exp3.env
move to exp3
evaluate exp1.type
evaluate exp1.opn
evaluate exp2.reqtype
evaluate exp3.reqtype
```

Question: Are there grammars for which we need to visit a node more than once?

Computation of Attribute Evaluation Order – Method 3

The final method for computing the order of attribute evaluation is for the system to order attribution evaluation based on the order in which productions are used in parsing. This is what `bison` provides. (Ullman et al) calls this the “oblivious method.”)

Clearly this is less general than the other two methods.

With a LR-parser we can restrict our attribute grammar so that

- Inherited attributes in a production $P : X \rightarrow X_1, \dots, X_n$ for a non-terminal X_i are computed from
 1. inherited attributes of X ,
 2. synthesized attributes of the symbols X_1, \dots, X_{i-1} (where attributes of terminal symbols are considered synthesized)
- Synthesized attributes in a production $P : X \rightarrow X_1, \dots, X_n$ for a non-terminal X are computed from
 1. inherited attributes of X ,
 2. synthesized attributes of symbols on the right-hand side of P ,

We can then perform attribute evaluation as handles are recognised and as symbols are expected.

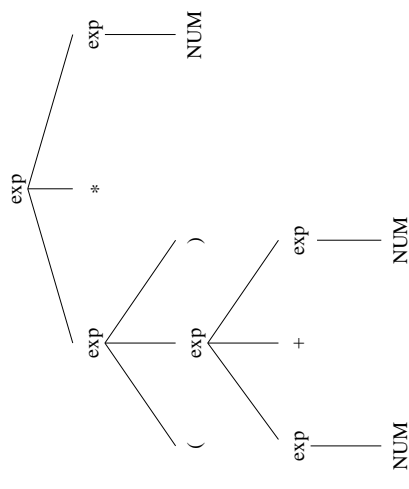
Construction of Structure Trees

Earlier we noted that usually the parser does not build the full parse tree but rather strips it to the essential **structure tree**, (sometimes called an **abstract syntax tree**.) This process can be specified using attributes.

Production	Semantic Rules
$exp_0 \rightarrow exp_1 + exp_2$	$exp_0.nptr := mknode('+', exp_1.nptr, exp_2.nptr)$
$exp_0 \rightarrow exp_1 - exp_2$	$exp_0.nptr := mknode('-', exp_1.nptr, exp_2.nptr)$
$exp_0 \rightarrow exp_1 * exp_2$	$exp_0.nptr := mknode('*', exp_1.nptr, exp_2.nptr)$
$exp_0 \rightarrow exp_1 / exp_2$	$exp_0.nptr := mknode('/', exp_1.nptr, exp_2.nptr)$
$exp_0 \rightarrow (exp_1)$	$exp_0.nptr := exp_1.nptr$
$exp_0 \rightarrow NUM$	$exp_0.nptr := mkleaf(NUM.val)$

The function *mkleaf* makes a leaf node in the tree while *mknode* makes a new node which points to the trees in its arguments.

For example, evaluation of the parse tree $(2 + 4) * 3$ leads to the abstract syntax tree:



Syntax-directed Translation

If the mapping of the source language to the target language is comparatively simple we can employ **syntax-directed translation**.

Syntax-directed translation consist of two phases:

1. build the **syntax-tree or structure tree during parsing using an attribute grammar**,
2. **traverse the syntax-tree recursively generating the target code.**

In very simple cases there is **no need to split the translation into two separate phases and syntax-directed translation can be performed as part of attribute evaluation.**

Keep in mind that even for very simple languages this renders the parser sensitive to changes in the target language.

Generation of RPN Instructions

As an example of simple syntax-directed translation imagine that we have to translate arithmetic expressions written in the usual infix form into instructions in **Reverse Polish Notation (RPN)** for evaluating the expression.

RPN instructions have form:

- $push(n)$ which pushes the number n onto the stack
- op which pops the required number of elements for the arithmetic operation op from the top of the stack, performs the operation on these elements and then pops the result back on to the top of the stack.

The expression $10+(3*4)$ is evaluated using the RPN instructions $push(10) push(3) push(4) * +$

We can use an attribute grammar to generate the RPN code for evaluating an expression. Some example productions are given below where @ concatenates two strings.

Production	Semantic Rules
$exp_0 \rightarrow exp_1 + exp_2$	$exp_0.code := exp_1.code@exp_2.code@"+"$
$exp_0 \rightarrow exp_1 - exp_2$	$exp_0.code := exp_1.code@exp_2.code@"-"$
$exp_0 \rightarrow (exp_1)$	$exp_0.code := exp_1.code$
$exp_0 \rightarrow NUM$	$exp_0.code := "push("@IntToString(NUM.val)@"")"$

Attribute Grammar Generators

There are many tools available for automatically generating a semantic analyzer from an attribute grammar. These include

- **Elegant.** The elegant system. Philips Research.
- **Eli.** The eli system, compiler construction made easy. University of Colorado at Boulder, University of Paderborn, Macquarie University in Sydney.
- **FNC-2.** The fnc-2 attribute grammars system. Didier Parigot, Oscar project, INRIA Rocquencourt.
- **FUN.** The fun transformation system. Attribute Grammar Based Transformation Systems.

Search the Web if you are interested. A good starting point is:

<http://www-rocq.inria.fr/oscar/www/fnc2/attribute-grammar-people.html>

Summary

We have looked at semantic analysis and attribute grammars.

Homework

- Read Chapter 5 of Aho et al.
- The reason for using attribute grammars is that have a higher expressive power than normal context-free grammars. We know that $L = a^n b^m c^n d^m$ is not a context-free language. Write an attribute grammar for L .
- Give the missing attribute rules and conditions for the other productions in the grammar for the desk top calculator language. Now use this to type check the program

```
real x;  
x := 100 + 8.9;
```

- Give a visit sequence for the production
 $assign \rightarrow IDENT := exp$
- Extend the large example and give attribute rules and conditions for the production
 $exp \rightarrow floor(exp)$,
where $floor$ takes a real and returns an integer.

Programming Language Implementation IX

In this lecture we will continue to look at **semantic analysis** and in particular at

- **Type inference**
- **Program optimisation**

The material is (very loosely) based on Mitchell Section 6.3 and 7.3.4 and Aho et al, Chapters 6 and 10 and Wilhelm and Maurer Chapter 10.

Type Checking and Type Inference

The semantic analysis phase of compilation checks non-syntactic properties of the program. **Type checking** and **type inference** are among the most important tasks of the semantic analysis.

In the last lecture we saw how attribute grammars can be used to perform type checking:

Grammar rule: $exp1 \rightarrow exp2 + exp3$

Attribution Rules:

```
exp2.env := exp1.env
exp3.env := exp1.env
exp1.type :=
  if exp2.type = int and exp3.type = int
  then int else real
exp1.opn :=
  if exp1.type = int then int_add else real_add
exp2.reqtype := exp1.type
exp3.reqtype := exp1.type
```

Note the type of the expression is determined from the types of the operands: information on the types flows in only in one direction (as long as the environment is only used for lookup).

Such a computation is only possible if the types of all identifiers and procedures are declared explicitly.

Thus attribute grammars can only perform **type checking**.

Type Inference

Consider the polymorphic ML function

```
fun len [] = 0
  | len (x::xs) = 1 + len xs;
```

ML will determine the type to

```
len: 'a list -> int
```

To determine such implicit types requires type information to be propagated in a much more complex way. As a consequence type inference is considerably harder than type checking.

We will now look at how ML performs type inference.

Hindley-Milner Type Inference

The **Hindley-Milner type system** is the basis for type inference in functional programming languages like ML, Objective Caml, Haskell etc and logic programming languages like Mercury and HAL. In theory it would also work with imperative and object oriented languages, it is just that as far as I know no-one has tried.

It returns a single most general type (well almost..) called the **principle type**.

Type Expressions

A type expression T is defined recursively to be

- a type variables r, s, t, u, v, w, \dots
- a simple type $bool, int, \dots$
- a type constructor with a type expression as its arguments, e.g. $list(T)$
- a function $T \rightarrow T$
- a tuple $T \times T$.

Note that for simplicity we use $a, b, c, \dots, r, s, t, u, v, w, \dots$ rather than the ML notation $'a, 'b, \dots$ for type variables

For instance, the ML type

```
len: 'a list -> int
```

corresponds to the expression

```
list(r) -> int
```

Note that the name chosen for a type variable does not matter.

Core ML

We will use a language *CoreML* to discuss type inference. It represents the core of ML-like languages and (most) ML functions can be translated into Core ML.

For simplicity we ignore mutually recursive functions, let declarations, equality types and overloading.

A program in Core ML is of form

```
val expr = expr
```

while expressions *expr* are defined recursively by

```
var          variable or identifier
fn var=> expr function definition
expr(expr)  function application
(expr, expr) tuple construction
case expr of pat=> expr | ... | pat=> expr case construct
```

For example the ML program

```
fun len [] = 0
  | len (x::xs) = 1 + len xs;
```

is equivalent to the Core ML program

```
val len = fn y => case y of [] => 0
          | (x::xs) => 1 + len(xs);
```

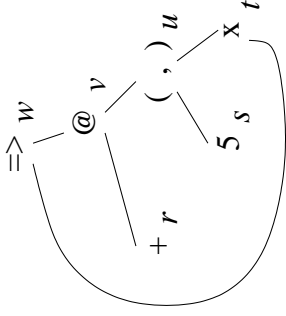
Note that all expressions are implicitly typed.

Simple Example

We can infer the type of the Core ML expression

`fn x => 5 + x`

as follows.



1) We assign a type variable to the expression and each subexpression in it.

We do this by constructing the parse tree for the expression. We then associate a different type variable with each node in the parse tree.

Note that we create a node labelled with @ for each function application. This represents the result of the application.

Simple Example (Cont.)

2) We generate a set of type equations (or constraints) on the type variables based on the parse tree.

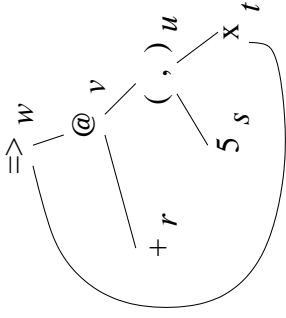
The kind of constraint generated depends on the form of the subexpression. In our example we use the rules

- **Function application** $f(e)$:
If the type of e is r and the type of the result is s then f must have type $r \rightarrow s$.
- **Function definition** $fn\ x=> e$:
If the type of e is s and the type of x is r then the expression $fn\ x=> e$ must have type $r \rightarrow s$.
- **Tuple creation** (e_1, e_2) :
If the type of e_1 is r and the type of e_2 is s then the expression (e_1, e_2) must have type $r \times s$.

We also add known type information for functions, constants and previously defined variables. In this case we know that

- the type of `+` is $int \times int \rightarrow int$ and
- `5` has type int .

Simple Example (Cont.)



For our example we generate the equations

$r = \text{int} \times \text{int} \rightarrow \text{int}$ known type
 $s = \text{int}$ known type
 $u = s \times t$ tuple creation
 $r = u \rightarrow v$ function application
 $w = t \rightarrow v$ function definition

Simple Example (Cont.)

3) We now solve the type equations

We simply eliminate variables from the equations until we reach a solved form. The order of elimination does not matter.

We start with the equations

$$r = \text{int} \times \text{int} \rightarrow \text{int}, s = \text{int}, u = s \times t, r = u \rightarrow v, w = t \rightarrow v$$

We can use $r = \text{int} \times \text{int} \rightarrow \text{int}$ to eliminate r from the other equations giving

$$r = \text{int} \times \text{int} \rightarrow \text{int},$$

$$s = \text{int}, u = s \times t, \text{int} \times \text{int} \rightarrow \text{int} = u \rightarrow v, w = t \rightarrow v$$

We can use $s = \text{int}$ to eliminate s from the other equations giving

$$r = \text{int} \times \text{int} \rightarrow \text{int}, s = \text{int},$$

$$u = \text{int} \times t, \text{int} \times \text{int} \rightarrow \text{int} = u \rightarrow v, w = t \rightarrow v$$

Now we can use $u = \text{int} \times t$ to eliminate u , giving

$$r = \text{int} \times \text{int} \rightarrow \text{int}, s = \text{int}, u = \text{int} \times t,$$

$$\text{int} \times \text{int} \rightarrow \text{int} = \text{int} \times t \rightarrow v, w = t \rightarrow v$$

Now we simplify the equation

$$int \times int \rightarrow int = int \times t \rightarrow v$$

It is equivalent to the equations $t = int, v = int$ so we can replace it by these giving

$$\begin{aligned} r &= int \times int \rightarrow int, s = int, u = int \times t, \\ t &= int, v = int, w = t \rightarrow v \end{aligned}$$

We can now use $t = int$ to eliminate t from the other equations . This gives

$$\begin{aligned} r &= int \times int \rightarrow int, s = int, u = int \times int, t = int, \\ v &= int, w = int \rightarrow v \end{aligned}$$

Finally we can use $v = int$ to eliminate v . This gives

$$\begin{aligned} r &= int \times int \rightarrow int, s = int, u = int \times int, t = int, \\ v &= int, w = int \rightarrow int \end{aligned}$$

Thus the type of the expression

```
fn x => 5 + x
```

is $int \rightarrow int$.

Algorithm for Type Inference

To summarise the algorithm is

- (1) We assign a type variable to the expression and each subexpression.
- (2) We generate a set of type equations (or constraints) on the type variables based on the parse tree and the known types
- (3) We solve the type equations

We now look at step (2) and (3) in more detail.

Type Rules

The rules for creating constraints from each sub-expression are:

- **Function application** $f(e)$
If the type of e is r and the type of the result is s then f must have type $r \rightarrow s$.
- **Function definition** $f n. x \Rightarrow e$
If the type of e is s and the type of x is r then the expression $f n. x \Rightarrow e$ must have type $r \rightarrow s$.
- **Tuple creation** (e_1, e_2)
If the type of e_1 is r and the type of e_2 is s then the expression (e_1, e_2) must have type $r \times s$.
- **Program** $val e_1 = e_2$
The type of e_1 and e_2 must be equal.

37

Type Rules (Cont.)

The last rule handles pattern matching

- **case** e of $p_1 \Rightarrow e_1 \mid \dots \mid p_n \Rightarrow e_n$
We treat each $p_i \Rightarrow e_i$ as if it defines a function:
Thus if the type of p_i is s_i and the type of e_i is t_i then the type of $p_i \Rightarrow e_i$ must be equal to $s_i \rightarrow t_i$.
We treat the *case* as if it applies the function defined by each $p_i \Rightarrow e_i$ to e .
Thus for each $p_i \Rightarrow e_i$ with type $s_i \rightarrow t_i$ we have that the type of e must be equal to s_i and the type of the result equal to t_i .

This relies on treating data constructors as if they are functions from their arguments to the data type they are constructors for.

For example consider the `list` data type

```
datatype 'a list =  
  [] |  
  :: of 'a * 'a list;
```

We must treat `[]` as a function with no arguments (i.e. a constant) with type $list(a)$ and `::` is a function with type $a \times list(a) \rightarrow list(a)$.

38

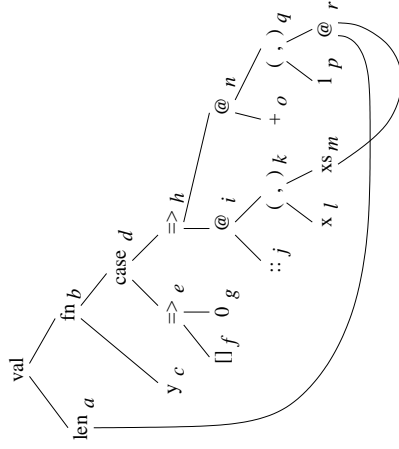
Another Example

```

val len = fn y =>
  case y of [] => 0
  | (x::xs) => 1 + len(xs);

```

(1) We assign a type variable to the expression and each subexpression:



Another Example (Cont.)

(2) We generate a set of type equations (or constraints) on the type variables based on the parse tree and the known types

$f = list(s)$	known type
$g = int$	known type
$j = t \times list(t) \rightarrow list(t)$	known type
$o = int \times int \rightarrow int$	known type
$p = int$	known type
$k = l \times m$	tuple creation
$q = p \times r$	tuple creation
$j = k \rightarrow i$	function application
$o = q \rightarrow n$	function application
$a = m \rightarrow r$	function application
$b = c \rightarrow d$	function definition
$e = f \rightarrow g$	$=_i$ construct in case
$h = i \rightarrow n$	$=_i$ construct in case
$e = c \rightarrow d$	case definition
$h = c \rightarrow d$	case definition
$a = b \rightarrow rd$	val

Another Example (Cont.)

(3) We solve the type equations

Left as an exercise!

Solving Type Equations

We simply

(1) eliminate variables from the equations

(2) simplify equations

until we reach a solved form. The order of processing does not matter.

This is similar to how systems of simultaneous linear equations can be solved using Gauss-Jordan elimination.

The actual algorithm that we use is called the **unification algorithm**.

Unification Algorithm

The Unification Algorithm takes some type equations C .
Returns *false* if C is unsatisfiable, otherwise returns the solved form of C .

```
 $S := true$ 
while  $C$  is not empty
  select equation  $c$  from  $C$ 
  if  $c$  is of the form  $x = x$ 
    (1) remove  $c$  from  $C$ 
  elseif  $c$  is of the form  $f(s_1, \dots, s_n) = f(t_1, \dots, t_n)$ 
    (2) replace  $c$  in  $C$  with  $s_1 = t_1, \dots, s_n = t_n$ 
  elseif  $c$  is of the form  $f(s_1, \dots, s_n) = g(t_1, \dots, t_m)$ 
    (3) return false
  elseif  $c$  is of the form  $t = x$  and  $t$  is not a variable
    (4) replace  $c$  by  $x = t$ 
  elseif  $c$  is of the form  $x = t$ 
    (5) if  $t$  contains  $x$  then return false
    (6) remove  $c$  from  $C$ 
    substitute  $t$  for  $x$  throughout  $C$  and  $S$ 
    add  $c$  to  $S$ 
  endif
endwhile
return  $S$ 

-  $x$  is a type variable,
-  $s, t, s_i, t_j$  are type expressions,
-  $c$  is a type constraint,
-  $C$  and  $S$  are sets of type constraints.
```

43

Example of the Unification Algorithm

Solve the set of type constraints

```
 $f = list(s)$  known type
 $g = int$  known type
 $b = c \rightarrow d$  function definition
 $e = f \rightarrow g =_l$  construct in case
 $h = i \rightarrow n =_l$  construct in case
 $e = c \rightarrow d$  case definition
 $h = c \rightarrow d$  case definition
```

44

Handling Polymorphism

Notice that we only create one node for each variable occurring in the program or expression and then have multiple arcs to that node if it occurs more than once in the expression.

Notice how recursive definitions are handled. If the variable v being defined occurs in the expression, it is assumed to have the same type everywhere.

However, this does not work for identifiers which refer to polymorphic functions and constants. In this case we need a different node for each occurrence of the identifier and we need to rename the type variables in the type apart.

Machine-Independent Optimization

Semantic analysis is also important in the machine-independent code optimisation phase. It may be used to do more **checking** and to determine when **efficiency increasing transformations** on the level of the source program code are applicable.

One example of checking is to determine that on every possible execution path each variable is initialized before being used.

Machine-independent code optimisation may be done at the level of the high-level language or at the intermediate code level, whichever is easier.

Typical code optimizations include:

- Calculation of sub-expressions whose values are already known.
- Extraction of loop-invariant computation from loops.
- Elimination of dead code.
- Procedure call inlining.
- Tail-recursion optimization.

Simple Optimization Example

Consider the C code

```
max = 100;
for(i=1; i++; i < max)
  for(j=i; j++; j < max) {
    x = 2*b[i];
    a[i,j] = x+j; }
```

We can optimise this code by

(1) Calculating sub-expressions whose values are already known. To do this we need to know which values are known at compile-time.

In the example `max` is: This gives

```
max = 100;
for(i=1; i++; i < 100)
  for(j=i; j++; j < 100) {
    x = 2*b[i];
    a[i,j] = x+j; }
```

47

Simple Optimization Example (Cont)

```
max = 100;
for(i=1; i++; i < 100)
  for(j=i; j++; j < 100) {
    x = 2*b[i];
    a[i,j] = x+j; }
```

(2) Extraction of loop-invariant computation from loops. Need to understand dependencies between variables and the loop control variables.

In the example `x` only depends on `i` so it can be moved to the outer loop: This gives

```
max = 100;
for(i=1; i++; i < 100) {
  x = 2*b[i];
  for(j=i; j++; j < 100) {
    a[i,j] = x+j; }}
```

Note this can be tricky—need to understand aliasing. Sometimes compilers are optimistic!

48

Simple Optimization Example (Cont)

```
max = 100;
for(i=1; i++; i < 100) {
    x = 2*b[i];
    for(j=i; j++; j < 100) {
        a[i,j] = x+j; }}
```

(3) Elimination of dead code and dead variables.

Need to know which variables are dead and which code can't be reached.

In the example `max` is never used so the initial assignment to `max` can be eliminated: This gives

```
for(i=1; i++; i < 100) {
    x = 2*b[i];
    for(j=i; j++; j < 100) {
        a[i,j] = x+j; }}
```

Note this is also tricky—again we need to understand aliasing.

Tail Recursion Optimisation

Tail recursion optimisation removes the overhead of a certain type of recursion. It is an important optimisation for functional and logic programming languages.

A call in the definition of function f to function g is a **tail call** if f returns the result of calling g without any other work.

For example the first call to f is a tail call while the second is not.

```
fun g(x) = if x=0 then f(x) else f(x)*2;
```

A function f is **tail recursive** if all of the recursive calls in the definition of f are tail calls.

Tail Recursion Optimisation

Tail recursion optimisation removes the recursion from tail recursive programs by essentially converting them to iteration.

Consider the ML program

```
fun tlfact(n,a) = if n <= 1 then a else tlfact(n-1,n*a);
```

This is tail recursive.

It is optimised to

```
fun tlfact(n,a) =  
  while not !n <= 1 do  
    (a := !n * !a; n := !n - 1);  
  !a;
```

Programs which use accumulators are usually tail recursive.

Another way of understanding tail recursive optimisation is that it reuses the same activation record on the stack—see next lecture.

Summary

We have looked at type inference and common code optimisations.

Homework

- Work out the type of

```
fun fst (x,y) = x;
```

and

```
fun fst_fst z = fst (fst z);
```

- Work out the type of

```
fun twice f x = f(f(x));
```

- Can tail recursion optimisation be used with the linear time version of reverse? If so give the equivalent ML code after the optimisation has been applied.

Programming Language Implementation X

In this lecture we will look at **code generation** and in particular at

- **runtime environment**
- **intermediate code generation**
- **register allocation**

The material is (loosely) based on Aho et al Chapters 7, 8 and 9 with some material from Wilhelm & Maurer Sections 5.2 and 5.3.

Run-time environment

To understand code generation one needs to understand what should happen at run-time. In particular we need to understand **allocation and deallocation** of data objects. This is managed by the **run-time support package**.

The design of the **run-time support package** is influenced by the HL language it supports.

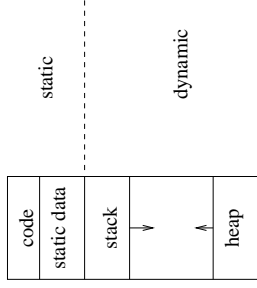
FORTRAN: The size of data structures is determined at compile time and sub-programs cannot be recursive.

Pascal family: Storage may be allocated dynamically and procedures can be passed as parameters and called recursively.

Functional and Logic languages: **Allocation and deallocation of storage is invisible to the programmer.**

We shall focus on the Pascal family (which includes C) but also look at compilation of object-oriented languages.

Activation Records



Memory is split into

- program code
- static data
- stack
- heap.

An activation record is pushed on to the stack when a procedure is called and popped off when control returns from the procedure.

The size of the activation record depends on the procedure's parameters and local variables.

Activation Records (Cont)

result if function
dynamic link
static link
return address
saved environment
parameters
local variables
temporary variables

The activation record might contain:

- A location for the result in the case it is a function.
- A pointer to the stack frame of the calling procedure (ie the **dynamic predecessor**).
- A pointer to the stack frame of the textually surrounding procedure (ie the **static predecessor**).
- The return address for the calling procedure.
- Environment information such as register values which need to be restored on return.
- Memory locations for the parameters.
- Memory locations for the local variables.
- Memory locations for the temporary variables generated in expression evaluation.

Normally we keep the **stack top (ST)** and the **local base (LB)** in registers.

Procedure Invocation

Calling a procedure/function

- Evaluates arguments and assigns values to the parameters.
- Sets the link data (ie dynamic and static predecessor and the return address.
- Jumps to procedure entry (found in some static table).

Returning from a procedure/function

- Restore the calling environment.
- Jump to the return address.
- If there is a return value, copy to temporary variable.

57

Static Links

The obvious question is why do we need a static link as well as a dynamic link?

Give the stack for the Pascal program:

```
program h {
  int i, j;

  int function p {
    int x;

    int function r {
      int y;
      j := j+1;
    }

    if i>0 then {
      i:= i-1;
      p();
    }
    else {
      r();
      rr: }
    }
  }

  i := 2;
  j := 1;
  p();
  p2r: }
```

58

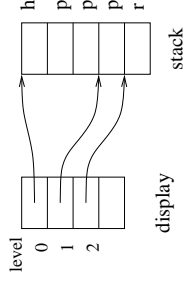
Static Links (Cont.)

The **textual level** of a procedure (block) is the number of procedures surrounding it.

In a Pascal-like language a procedure at level K can only call procedure at level $\leq K + 1$.

To access a variable at offset x level j from a procedure at level i we follow $i - j$ static links up the stack to find the right activation record and then go to off set x .

Displays



We can avoid the unravelling of static links by maintaining an array of registers indexed by textual level which point to the appropriate activation record.

This is called a **display**.

If i is the level of the called procedure we need to set $display[i]$ on call and restore its value on return from the call.

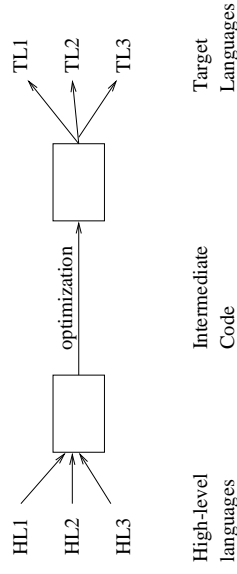
Code Generation

The aim is to produce code which is:

- efficient
- compact
- uses registers wisely
- is correct.

Typical steps in code generation are:

- Intermediate code generation
- Optimization (this is optional)
- Target code generation



Using a machine-independent intermediate language is good because:

- retargeting is facilitated
- facilitates machine-independent code optimization

Intermediate Code

One common form of intermediate code is **three address code**. This is a sequence of statements of form

$x := y <op> z$

Thus a complex source language expression like

$x := y + u * z$

will give rise to the statements

$t1 := u * z$
 $x := y + t1$

Three address statements are similar to assembly code:

- **Assignment statements of the above form.**
- **Assignment statements of the form**

$x := <op> z$

where $<op>$ is a unary operation.

- **Copy statements of the form**

$x := z$

- **Unconditional jumps of form**

`goto L`

- **Conditional jumps of form**

`if x <relop> y goto L`

- **Parameter setting statements and procedure call and return y.**

```
param x1
param x2
...
param xn
call p, n
```

- **Indexed assignments of form**

```
x := y[i]
y[i] := x
```

- **Address and pointer assignments of form**

```
x := &y
x := *y
*x := y
```

This is only one possible intermediate code. See Chapter 8 of Aho et al for more details.

Intermediate Code Generation

It is straightforward to use attribute grammars to construct intermediate code.

Production	Semantic Rules
$assign \rightarrow id := exp$	$assign.code ::=$ $exp.code \oplus gen(id.place := exp.place)$
$exp_0 \rightarrow exp_1 + exp_2$	$exp_0.place ::= newtemp$ $exp_0.code ::=$ $exp_1.code \oplus exp_2.code \oplus$ $gen(exp_0.place := exp_1.place + exp_2.place)$
$exp_0 \rightarrow exp_1 * exp_2$	$exp_0.place ::= newtemp$ $exp_0.code ::=$ $exp_1.code \oplus exp_2.code \oplus$ $gen(exp_0.place := exp_1.place * exp_2.place)$
$exp_0 \rightarrow (exp_1)$	$exp_0.place := exp_1.place$ $exp_0.code := exp_1.code$
$exp_0 \rightarrow id$	$exp_0.place := id.place$ $exp_0.code := nil$

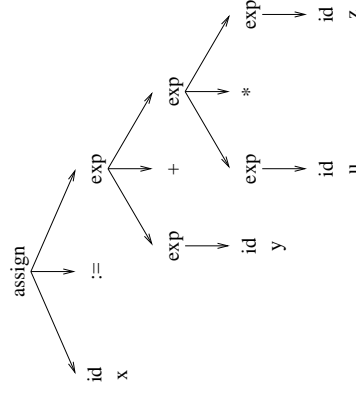
Intermediate Code Generation (Cont)

Abstract Machines / Virtual Machines

For example

$x := y + u * z$

will give rise to:



Instead of compiling directly into machine code, the target is often a **virtual machine** or **abstract machine**.

For instance Prolog is often compiled into the WAM (Warren Abstract Machine) code, and Java into JVM (Java Virtual Machine) code.

A virtual machine is essentially an interpreter for a virtual language that runs on the target machine.

The virtual language provides an intermediate level between high-level language and native machine code that is specifically designed for a particular class of languages. (e.g. procedural, object-oriented, logic, functional) and provides the basic data structures and basic control mechanisms in these languages.

Since the virtual machine language is closer to the high-level language, translation into virtual machine code is easier. Compilation into native machine code is more complex, since machine code requires explicit data structure and address management, provides only very simple forms of control structures and requires more optimization.

Abstract Machines / Virtual Machines (Cont.)

The main arguments for using a virtual machine (such as the JVM) are

- ease of implementation
- increased portability
- security and better run-time checking (easier encapsulation)

However, native code is typically much faster.

Virtual machine code is much like machine-independent intermediate code.

Target Code Generation

Instruction selection

If we do not care about efficiency it is usually easy to generate target code from each three address statement.

Unfortunately we usually do care about efficiency!

Register allocation

It is better to use register operands. Use of registers involves two steps

- **Register allocation** in which we select the variables to be stored in registers.
- **Register assignment** in which we select the specific registers.

We note that optimal assignment is NP-hard.

Register Allocation

In fact register allocation can be viewed as **k -graph colouring**.

In the graph, variables are nodes and nodes are connected if they are alive at the same time. If we have k registers we try to color the graph nodes so that no two connected nodes have the same color using only k colours.

Consider the intermediate code

```
<x,y alive>
t1 := x + 1
t2 := y + t1
t3 := y * t2
z := x + t3
<x,z alive>
```

What is the register allocation graph?

Compilation of Object-Oriented Languages

Object-oriented languages, such as C++, extend imperative programming languages by providing

- objects
- methods
- class hierarchy with inheritance

We now investigate how these can be translated into standard imperative language constructs.

What is the minimal number of registers needed?

Example

```
class Point {
    int x, y;
public:
    int getX();
    int getY();
    void move(int,int);
    virtual void draw();
protected: ...
private: ...
}
class ColourPoint: public Point {
public:
    int getX();
    int getY()
    void move(int,int);
    void draw();
    int getColour();
    void setColour(int);
protected: ...
private: ...
}
```

71

Compilation of Methods

Objects are translated into a structure which has a field for each data element in the class.

Each method of a class is translated into a procedure which takes the object itself as an extra argument.

For instance the method `move` defined by

```
void move(int nx, int ny) {
    x = nx;
    y = ny; }
```

will be translated into

```
void move(Point &this, int nx, int ny) {
    this.x = nx;
    this.y = ny; }
```

72

Compilation of Inheritance and Subtype Polymorphism

The hard part about compilation of object-oriented languages is supporting subtype polymorphism.

A common solution is for the compiler to create a **method table** for each class. This contains pointers to the functions implementing all virtual methods for that class.

The structure representing an object includes a pointer to the object's method table. Calls to methods which are virtual are made via this method table.

In our example above the method `draw` is virtual. A call to `pt.draw()` will be translated into

```
pt_struct.method_table[0](pt_struct);
```

where `pt_struct` is the structure corresponding to `pt` and the function implementing `draw` is at index 0 in the method table.

Note that the method tables for `Point` and `ColourPoint` will point to different functions implementing `draw`.

For more details see [Wilhelm & Maurer](#).

Summary

We have looked at

- runtime environment
- intermediate code generation
- register allocation
- compilation of object-oriented languages

Homework

- Read Chapters 7, 8 and 9 of Aho et al.
- How do you compute the value of the static link in the procedure being called?
- Give attribute rules to generate code for a while loop and a procedure call (first give the grammar!).
- Do the assignment!