# 5. Semantic Analysis II

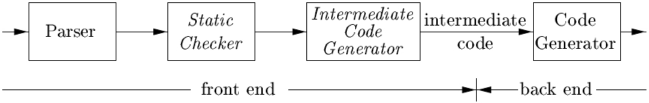
## Type Checking – Intermediate Code Generation

Andrea Polini, Luca Tesei

Formal Languages and Compilers
MSc in Computer Science
University of Camerino

# ToC

# Intermediate Code generation




- Last block in the front end of a compiler. We will consider:
  - intermediate representations – memory management is still abstracted
  - static checking – type checking in particular
  - intermediate code generation – the C programming language is often selected as an intermediate form because it is flexible, it compiles into efficient machine code and its compilers are widely available.

# Intermediate Representations

The two most important intermediate representations are:

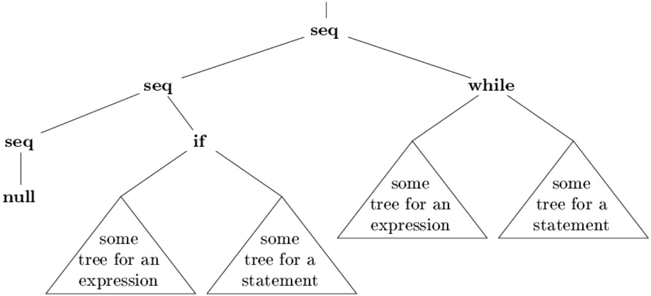
- Trees: parse trees, (abstract) syntax trees
- Linear representations: three-address code

# Tree Representations

- (Abstract) syntax trees can be generated during parsing using a synthesized attribute
- Let's see an example for a basic *while language*

# Tree Representations

- Example of syntax tree



# Concrete vs Abstract Syntax

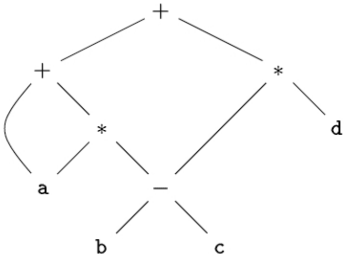
- Nodes with “similar” treatment for translation and type checking can be grouped

| CONCRETE SYNTAX | ABSTRACT SYNTAX |
|---|---|
| = | **assign** |
| \|\| | **cond** |
| && | **cond** |
| == != | **rel** |
| < <= >= > | **rel** |
| + - | **op** |
| * / % | **op** |
| ! | **not** |
| $-_{unary}$ | **minus** |
| [] | **access** |

# Direct Acyclic Graph (DAG)

A Direct Acyclic Graph (DAG) can be considered a compacted form of an Abstract Syntax Tree where common terms are not repeated. The result is that “leaves” will have more than one parent resulting in a graph rather than a tree structure

Consider the case of the expression $a + a * (b - c) + (b - c) * d$



# DAG generation

**How to generate it**

The derivation of a DAG is much similar to that of a AST. In particular it is enough to revise the implementation of the `Node` method to avoid the replications of nodes

Let's recall the SDD for simple expressions...

# Three Address Code

The term “three-address code” comes from instructions of the general form $x = y \ op \ z$ with three addresses (two for the operands and one for the result

In “three-address code” operations there is at most one operator on the right side of each single instruction.
Consider the expression: `x+y*z` the codification will look like ...

**Building blocks**

Three address code is built from two concepts: addresses, instructions.

# Three Address Code

The term “three-address code” comes from instructions of the general form $x = y \; op \; z$ with three addresses (two for the operands and one for the result

In “three-address code” operations there is at most one operator on the right side of each single instruction.
Consider the expression: `x+y*z` the codification will look like . . .

**Building blocks**

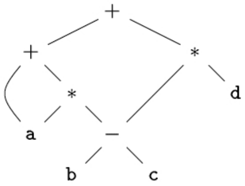
Three address code is built from two concepts: addresses, instructions.

# Founding concepts

## Addresses

- name
- constant
- compiler generated temporary

Three address codes are linearized representation of a syntax tree or DAG




(a) DAG

```
t1 = b - c
t2 = a * t1
t3 = a + t2
t4 = t1 * d
t5 = t3 + t4
```

(b) Three-address code

# Founding concepts

## Instructions

- assignment (with binary and unary operators) – e.g. $x = y\ op\ z$, $x = op\ y$
- copy instructions – e.g. $x = y$
- unconditional jump – e.g. `goto` $L$
- conditional jump with boolean – `if` $x$ `goto` $L$, `ifFalse` $x$ `goto` $L$
- conditional jump with relational operators – `if` $x\ relop\ y$ `goto` $L$
- procedure calls and returns – e.g. `param x`, `call p,n`, and `y = call p,n`
- indexed copy instructions – e.g. `x=y[i]` and `x[i]=y`)
- Address and pointer assignment – e.g. `x=&y`, `x=*y`, `*x=y`)

# Three address code representation and storage

Let's provide a translation for the following code fragment:

```
do
  i=i+1;
while (a[i] < v);
```

Data structures to represent the intermediate code (see book):

- Quadruples – includes results
- Triples
- Indirect Triples
- Static Single-Assignment (SSA) form

# Three address code representation and storage

Quadruples example:

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

| | *op* | $arg_1$ | $arg_2$ | *result* |
|---|---|---|---|---|
| 0 | minus | c | | $t_1$ |
| 1 | * | b | $t_1$ | $t_2$ |
| 2 | minus | c | | $t_3$ |
| 3 | * | b | $t_3$ | $t_4$ |
| 4 | + | $t_2$ | $t_4$ | $t_5$ |
| 5 | = | $t_5$ | | a |
| | ... | | | |

(b) Quadruples

# ToC

# Types and Declarations

Types establish sets in which program elements can get their values.
Two main activities related to compiling:

- Type Checking uses logical rules to reason about the behaviour of program at run time
- Translation Applications in which type related information are useful to determine the memory space needed for names at run-time, to compute address denoted by array reference, to apply conversions, to determine the operators to apply ...

# Type Expression

## Type Expressions

A type expression is either a basic type of is formed by applying an operator, called type constructor, to a type expression. E.g. `int[2][3]`

## inductive constructions of types expressions

- A basic type is a type expression (generally languages include basic types such as – boolean, char, integer, float, void, double, . . . )
- A type name is a type expression
- The array operator can be applied to a type expression to form a new type expression
- A record form a type expression from a list of type expressions
- The function operator ($\rightarrow$) can be used to define a function from a type $s$ to type $t$
- The Cartesian product for two type expressions results in a new type expression

# Declarations

Let's consider a simplified grammar for declarations:

$$D \rightarrow T\ \mathbf{id};\ D \mid \epsilon \quad T \rightarrow BC \mid \mathbf{record}\ '\{'\ D\ '\}'$$

$$B \rightarrow \mathbf{int} \mid \mathbf{float} \quad C \rightarrow \epsilon \mid [\mathbf{num}]C$$

# Types and storage allocation

Worth to be mentioned:

- Relative addresses can be assigned at compile time
- Addressing constraints of the target machine influence assignment of addresses

# Types and storage allocation for sequence of declarations

| | | | | |
|---|---|---|---|---|
| $P$ | $\rightarrow$ | | { | offset=0} |
| | | $D$ | | |
| $D$ | $\rightarrow$ | $T$ **id**; | { | top.put(**id**.lexeme,T.type,offset);<br>offset = offset + T.width; } |
| | | $D_1$ | | |
| | \| | $\epsilon$ | | |
| $T$ | $\rightarrow$ | $B$ | { | t=B.type; w=B.width; } |
| | | $C$ | { | T.type = C.type; T.width = C.width; } |
| | \| | **record** $'\{'$ | { | Env.push(top); top = new Env();<br>Stack.push(offset); offset=0; } |
| | | $D\ '\}'$ | { | T.type=record(top); T.width=offset;<br>top=Env.pop(); offset=Stack.pop(); } |
| $B$ | $\rightarrow$ | **int** | { | B.type=integer; B.width=4; } |
| | \| | **float** | { | B.type=float; B.width=8; } |
| $C$ | $\rightarrow$ | [**num**]$C_1$ | { | array(num.value,$C_1$.type);<br>C.width=num.value $\times$ $C_1$.width; } |
| | \| | $\epsilon$ | { | C.type = t; C.width = w; } |

# Translation of Expressions

In the translation of an expression we need to represent the code for the expression and the address in which the computed value will be stored. Therefore let's consider an excerpt for the usual expression grammar:

$$S \to \mathbf{id} = E \quad E \to E_1 + E_2 | - E_1 | (E_1) | \mathbf{id}$$

# SDD for three address code translation

| | | | |
|---|---|---|---|
| $S$ | $\rightarrow$ | $\mathbf{id} = E$ | S.code=E.code \|\|<br>gen(top.get(**id**.lexeme) '=' E.addr) |
| $E$ | $\rightarrow$ | $E_1 + E_2$ | E.addr=**new** Temp()<br>E.code = $E_1$.code \|\| $E_2$.code \|\| gen(E.addr '=' $E_1$.addr '+' $E_2$.addr) |
| | \| | $-E_1$ | E.addr=**new** Temp()<br>E.code = $E_1$.code \|\| gen(E.addr '=' '**minus**' $E_1$.addr) |
| | \| | $(E_1)$ | E.addr= $E_1$.addr, Ecode =$E_1$.code |
| | \| | **id** | E.addr =top.get(**id**.lexeme), E.code= ' ' |

Consider the expression "a=b+-c" and derive the three address code translation applying the semantic rules defined

# Type Checking

Type Checking ensures that the type of a construct matches that expected by its content. In:

$$\textbf{if}\ (\ expr\ )\ stmt$$

the expression *expr* is expected to have type **boolean**

# Type Checking

- Type checking rules follow the operator/operand structure of the abstract syntax tree
- e.g. the operator **rel** represents the operator $\leq$ (and other relational ones such as $<$, $==$, $>$ and $\geq$)
- the type rule for **rel** says that the two operands must have the same type and the resulting type is **boolean**
- a synthesized attribute *type* may be used for implementing the rule in an SDD:

**if** ( $E_1.type == E_2.type$ ) $E.type = \mathrm{boolean}$;
**else** error;

# Type Checking

To do type checking is necessary to assign a type expression to each component of the source program. Then a set of logical rules (type system) are defined to check if any non conformity is spotted.

Type checking can take two forms:

- type synthesis
- type inference

# Type Synthesis

In type synthesis the type of an expression is derived from those of its sub-expressions. Names need to be declared before usage.
A typical rule will look like the following one:

***if*** *f has type* $s \rightarrow t$ ***and*** *x has type s,*
***then*** *expression* $f(x)$ *has type t*

e.g. consider the case of $E_1 + E_2$

# Type inference

With type inference the type of a construct is determined from the way it is used.
A typical rule for the type inference has the form:

> ***if*** *$f(x)$ is an expression,*
> ***then*** *for some $\alpha$ and $\beta$, $f$ has type $\alpha \to \beta$* ***and*** *$x$ has type $\alpha$*

# Type conversion

Consider the expression $a = b + c$ where the variable do not necessarily have the same type....managed via type conversion
All languages have their specific rules defined for conversion:

- narrowing
- widening

Conversion can be implicit or explicit. Implicit conversions, also called coercions, are generally limited to widening. Explicit conversions are consequence of statements included by the programmer (cast).

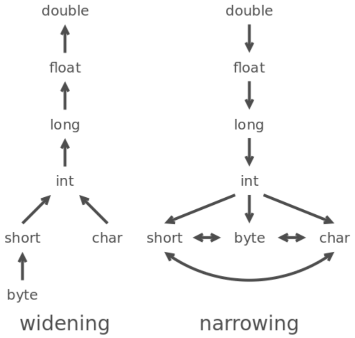
# Type conversion

Consider the expression $a = b + c$ where the variable do not necessarily have the same type....managed via type conversion

All languages have their specific rules defined for conversion:

- narrowing
- widening

Conversion can be implicit or explicit. Implicit conversions, also called coercions, are generally limited to widening. Explicit conversions are consequence of statements included by the programmer (cast).



# Type conversion

To define semantic actions for type checking two auxiliary functions are defined:

- *max*$(t_1, t_2)$: takes two types and return the maximum
- *widen*$(a, t, w)$ – where a is an address, while t and w are types: generate type conversion if needed to widen an address *a* of type *t* into a value of type *w*.

$E \rightarrow E_1 + E_2$ { E.type = max($E_1$.type,$E_2$.type);
a$_1$ = widen ($E_1$.addr,$E_1$.type,E.type)
a$_2$ = widen ($E_2$.addr,$E_2$.type,E.type)
E.addr = **new** Temp();
gen(E.addr '=' a$_1$ '+' a$_2$); }

# Type conversion

To define semantic actions for type checking two auxiliary functions are defined:

- *max*$(t_1, t_2)$: takes two types and return the maximum
- *widen*$(a, t, w)$ – where a is an address, while t and w are types: generate type conversion if needed to widen an address *a* of type *t* into a value of type *w*.

$$
\begin{array}{llll}
E & \rightarrow & E_1 + E_2 & \{ \quad \text{E.type = max(E}_1\text{.type,E}_2\text{.type);} \\
 & & & \quad\ \ \text{a}_1 = \text{widen (E}_1\text{.addr,E}_1\text{.type,E.type)} \\
 & & & \quad\ \ \text{a}_2 = \text{widen (E}_2\text{.addr,E}_2\text{.type,E.type)} \\
 & & & \quad\ \ \text{E.addr = } \textbf{new} \text{ Temp();} \\
 & & & \quad\ \ \text{gen(E.addr '=' a}_1 \text{ '+' a}_2\text{);} \quad \}
\end{array}
$$

# ToC

# Control Flow

Boolean expression are the building block for influencing the flow of a program. They are manipulated to:

- Alter the flow of control – like in `if` $(E)S$
- Compute logical values

Two different approaches to evaluation:

- Eager
- Lazy

# Short-Circuit Code

- Boolean operators ||, && and ! translate into jumps
- The operators do not appear on the code
- The value of a boolean expression is represented by a position in the code sequence

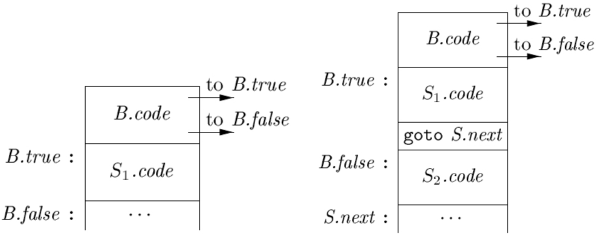
```
if (x < 100 || x > 200 && x != y) x = 0;
```

is translated to

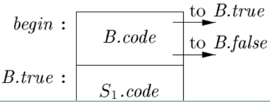
```
      if x < 100 goto L2
      ifFalse x > 200 goto L1
      ifFalse x != y goto L1
L2:   x = 0
L1:   ...
```

# Translation using jumping code

- Statements and Boolean Expressions have a synthesised attribute *code* that is a string containing the translated code
- Boolean Expressions have two inherited attributes *true* and *false* representing labels to which the control flows if the expression is true or false, respectively




(a) if (b) if-else



# Control Flow – commands

| | | | |
|---|---|---|---|
| $P$ | $\rightarrow$ | $S$ | S.next=newlabel(), P.code = S.code \|\| label(S.next) |
| $S$ | $\rightarrow$ | **assign** | S.code=**assign**.code |
| $S$ | $\rightarrow$ | **if** ($B$) $S_1$ | B.true=newlabel(), B.false=$S_1$.next=S.next<br>S.code=B.code\|\|label(B.true)\|\|$S_1$.code |
| $S$ | $\rightarrow$ | **if** ($B$) $S_1$ **else** $S_2$ | B.true=newlabel(), B.false=newlabel()<br>$S_1$.next=$S_2$.next=S.next<br>S.code=B.code\|\|label(B.true)\|\|$S_1$.code<br>\|\|gen('`goto`' S.next)\|\| label(B.false)\|\| $S_2$.code |
| $S$ | $\rightarrow$ | **while** ($B$) $S_1$ | begin=newlabel(), B.true=newlabel()<br>B.false=S.next, $S_1$.next=begin<br>S.code=label(begin)\|\|B.code\|\|label(B.true)\|\|$S_1$.code<br>\|\|gen('`goto`' begin) |
| $S$ | $\rightarrow$ | $S_1$ $S_2$ | $S_1$.next=newlabel(), $S_2$.next=S.next<br>S.code = $S_1$.code\|\|label($S_1$.next)\|\|$S_2$.code |

# Control Flow – Boolean expressions

| | | | |
|---|---|---|---|
| $B$ | $\rightarrow$ | $B_1 \| \| B_2$ | $B_1$.true=B.true<br>$B_1$.false=newlabel()<br>$B_2$.true = B.true<br>$B_2$.false=B.false<br>B.code = $B_1$.code \|\| label($B_1$.false)\|\| $B_2$.code |
| $B$ | $\rightarrow$ | $B_1 \&\& B_2$ | $B_1$.true=newlabel()<br>$B_1$.false=B.false<br>$B_2$.true = B.true<br>$B_2$.false=B.false<br>B.code = $B_1$.code \|\| label($B_1$.true)\|\| $B_2$.code |
| $B$ | $\rightarrow$ | $E_1$**rel**$E_2$ | B.code = $E_1$.code\|\|$E_2$.code<br>\|\| gen('`if`' $E_1$.addr **rel**.op $E_2$.addr '`goto`' B.true)<br>\|\| gen('`goto`' B.false) |
| $B$ | $\rightarrow$ | **true** | B.code=gen('`goto`' B.true) |
| $B$ | $\rightarrow$ | **false** | B.code=gen('`goto`' B.false) |

# Control Flow – commands

Let's translate the following program:

```
if (x != y && x == z) x = y + z;
```