5. Semantic Analysis II

Type Checking – Intermediate Code Generation

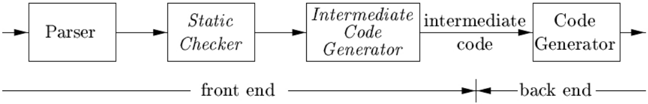
Andrea Polini, Luca Tesei

Formal Languages and Compilers
MSc in Computer Science
University of Camerino

ToC

1. Preliminaries
2. Types
3. Control Flow

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:

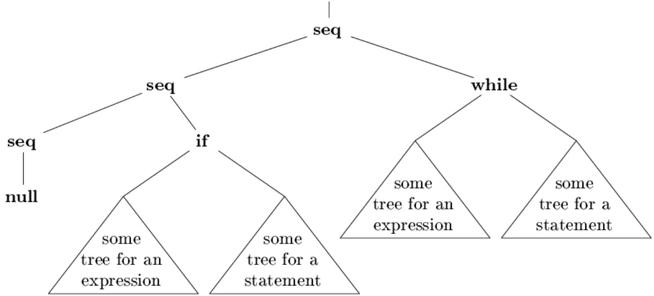
- Trees: parse trees, (abstract) syntax trees
- Linear representations: three-address code

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

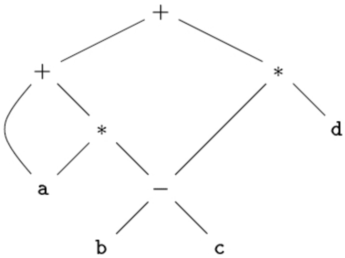
- Nodes with “similar” treatment for translation and type checking can be grouped

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

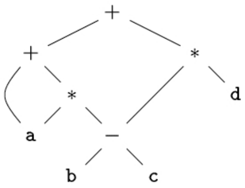
Three address code is built from two concepts: addresses, instructions.

Founding concepts

Addresses

- name
- constant
- compiler generated temporary

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



(a) DAG

$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = a + t_2$
$t_4 = t_1 * d$
$t_5 = t_3 + t_4$

(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:

$$
\begin{array}{rcl}
t_1 & = & \text{minus } c \\
t_2 & = & b * t_1 \\
t_3 & = & \text{minus } c \\
t_4 & = & b * t_3 \\
t_5 & = & t_2 + t_4 \\
a & = & t_5
\end{array}
$$

(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

1. Preliminaries
2. Types
3. Control Flow

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:

$$D \to T\ \mathbf{id};\ D \mid \epsilon \qquad T \to BC \mid \mathbf{record}\ '\{'\ D\ '\}'$$

$$B \to \mathbf{int} \mid \mathbf{float} \qquad C \to \epsilon \mid [\mathbf{num}]C$$

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); offset = offset + T.width; }
		D_1	
	$\mid$	ϵ	
T	$\rightarrow$	B	{ t=B.type; w=B.width; }
		C	{ T.type = C.type; T.width = C.width; }
	$\mid$	**record** $'\{'$	{ Env.push(top); top = new Env(); Stack.push(offset); offset=0; }
		$D\ '\}'$	{ T.type=record(top); T.width=offset; top=Env.pop(); offset=Stack.pop(); }
B	$\rightarrow$	**int**	{ B.type=integer; B.width=4; }
	$\mid$	**float**	{ B.type=float; B.width=8; }
C	$\rightarrow$	[**num**]C_1	{ array(num.value,C_1.type); C.width=num.value $\times$ C_1.width; }
	$\mid$	ϵ	{ 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 || gen(top.get(**id**.lexeme) '=' E.addr)
E	$\rightarrow$	$E_1 + E_2$	E.addr=**new** Temp() E.code = E_1.code || E_2.code || gen(E.addr '=' E_1.addr '+' E_2.addr)
	|	$-E_1$	E.addr=**new** Temp() 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