MIF08 : Compilation

Laure Gonnord

2016-2017

Compilation : reminder on LC-3 architecture

MIF08

Laure Gonnord
`Laure.Gonnord@univ-lyon1.fr`

Outline

Our target machine : LC-3

- memory: 2^{16} 16-bits words.
- instructions are also encoded in 16-bits words.

We have a simulator (Pennsim, see lab)

Registers **PC**, **IR**, **PSR** (*Program Status Register*) + 8 general purpose registers **R0,...,R7**.

LC-3 ISA

See companion document.

Example : ADD instruction

- `ADD DR, SR1, SR2`, does `DR <- SR1 + SR2`.
  - ↪ All operands are registers.
  - ↪ Example : `ADD R1, R2, R3` executes `R1 <- R2 + R3`.
- `ADD DR, SR1, imm5`, does `DR <- SR + imm5`.
  - ↪ The last operand is an immediate value encoded in the instruction.
  - ↪ Example : `ADD R1, R2, 5` executes `R1 <- R2 + 5`.

LC-3 ADD : encoding

A bit specifies the addressing mode:

assembly	**action**	**encoding**															
		opcode				arguments											
		F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
ADD DR,SR1,SR2	DR <- SR1 + SR2	0 0 0 1				DR			SR1			0	0 0		SR2		
ADD DR,SR1,Imm5	DR <- SR1 + Imm5	0 0 0 1				DR			SR1			1	Imm5				

LC-3 Memory access instructions

- `LD DR, add`, does `DR <- mem[add]`.
- `ST SR, add`, does `mem[add] <- SR`.

▶ **Direct addressing**: `add` is an address, `mem[add]` the associated memory cell.
This adress is encoded in the instruction:

assembly	**action**	**encoding**															
		opcode				arguments											
		F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
LD DR,add	DR <- mem[add]	0 0 1 0				DR			add								
ST SR,add	mem[add] <- SR	0 0 1 1				SR			add								

LC-3: branching

Unconditional branching:

- `BR add`, does `PC <- add`.

`add` denotes the memory address of the instruction that will be executed after the current instruction (direct addressing mode)

assembly	**action**	**encoding**															
		opcode				arguments											
		F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
BR add	PC <- add	0 0 0 0				0 0 0			add								

Outline

Ex : Assembly code - demo

```
        .ORIG x3000     ; address in memory
;program instructions
        LD R1,a         ;
                        ;
        LD R2,b         ;
                        ;
        ADD R0,R1,R2    ;
                        ;
        ST R0,r         ;
                        ;
        HALT            ; flow back to OS
; data
a:      .FILL #5        ; 5 is stored at this address
b:      .FILL #2        ;
r:      .BLKW #1        ; reservation of a memory cell
        .END
```

LC-3 Exercises

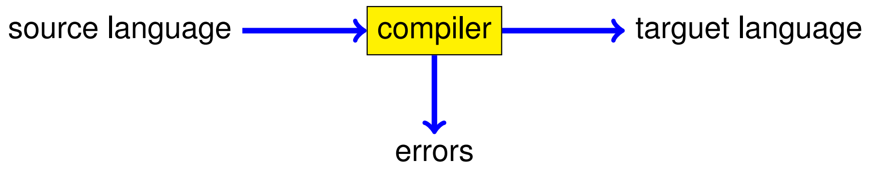
see TD sheet.

Introduction

MIF08

Laure Gonnord
Laure.Gonnord@univ-lyon1.fr

What's compilation?



Compilation toward the machine language

We immediatly think of the translation of a high-level language (C,Java,OCaml) into the machine language of a processor (Pentium, PowerPC. . .)

```
% gcc -o sum sum.c
```

```
int main(int argc, char **argv) {
  int i, s = 0;
  for (i = 0; i <= 100; i++) s += i*i;
  printf("0*0+...+100*100 = %d\n", s);
}
```

$\longrightarrow$

```
00100111101111011111111111110000010101111101111110000000000010100
10101111101001000000000000010000010101111101001010000000000100100
10101111101000000000000000001100010101111101000000000000000011100
10001111101011100000000000011100
```

But this is only one aspect, we will see more!

Course Objective

Be familiar with the mecanisms inside a (simple) compiler.
Beyond the scope: compilers optimisations of the 21$^{\text{th}}$ century.

Course Content

- Syntax Analysis : lexing, parsing, AST
- Evaluators
- Code generation
- (Code Optimisation)

Support language: Python 2.7
Frontend infrastructure : ANTLR 4.

Course Organization

- 4 TD groups: S. Brandel, L. Gonnord, N. Louvet, X.Urbain (`@univ-lyon1.fr`)
- 6 (or 7) TP groups: S. Brandel, T. Excoffier, E. Guillou, S. Guelton+Kevin Marquet, G. Bouchard, N. Louvet, X.Urbain.

The official URL :
`http://laure.gonnord.org/pro/teaching/compilM1.html`

Evaluation

- One “quick” during an exercise session (surprise!).
- Some of the lab exercises, 2 mini-projects.
- A final exam.

Syntax Analysis

MIF08

Laure Gonnord
`Laure.Gonnord@univ-lyon1.fr`

Goal of this chapter

- Understand the syntaxic structure of a language;
- Separate the different steps of syntax analysis;
- Be able to write a syntax analysis tool for a simple language;
- Remember: syntax$\neq$semantics.

Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens:
  - Words: groups of letters;
  - Punctuation;
  - Spaces.

Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.

Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.
- Group tokens into:
  - Propositions;
  - Sentences.

Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.
- Group tokens into: **Parsing**
  - Propositions;
  - Sentences.

Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.
- Group tokens into: **Parsing**
  - Propositions;
  - Sentences.
- Then proceed with word meanings:
  - Definition of each word.
    ex: a dog is a hairy mammal, that barks and...
  - Role in the phrase: verb, subject, ...

Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.
- Group tokens into: **Parsing**
  - Propositions;
  - Sentences.
- Then proceed with word meanings: **Semantics**
  - Definition of each word.
    ex: a dog is a hairy mammal, that barks and...
  - Role in the phrase: verb, subject, ...

Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.
- Group tokens into: **Parsing**
  - Propositions;
  - Sentences.
- Then proceed with word meanings: **Semantics**
  - Definition of each word.
    ex: a dog is a hairy mammal, that barks and...
  - Role in the phrase: verb, subject, ...

Syntax analysis=Lexical analysis+Parsing

Outline

What for ?

```
int y = 12 + 4*x ;
```

$\Longrightarrow$ [TINT, VAR("y"), EQ, INT(12), PLUS, INT(4), FOIS, VAR("x"), PVIRG]

▶ Group characters into a list of **tokens**, e.g.:
- The word “int” stands for *type integer*;
- A sequence of letters stands for a *variable*;
- A sequence of digits stands for an *integer*;
- ...

What’s behind

From a Regular Language, produce a Finite State Machine (see **LIF15**)

Tools: lexical analyzer constructors

- Lexical analyzer constructor: builds an automaton from a regular language definition;
- Ex: Lex (C), JFlex (Java), OCamllex, **ANTLR** (multi), ...
- **input**: a set of regular expressions with actions (`Toto.g4`);
- **output**: a file(s) (`Toto.java`) that contains the corresponding automaton.

Analyzing text with the compiled lexer

- The **input of the lexer** is a text file;
- Execution:
  - Checks that the input is accepted by the compiled automaton;
  - Executes some actions during the “automaton traversal”.

Lexing tool for Java: ANTLR

- The official webpage : `www.antlr.org` (BSD license);
- ANTLR is both a lexer and a parser;
- ANTLR is multi-language (not only Java).

▶ During the labs; we will use the Python back-end (here, demo in java)

ANTLR lexer format and compilation

.g4

```
grammar XX;

@header {
// Some init code...
}

@members {
// Some global variables
}
// More optional blocks are available

--->> lex rules
```

Compilation:

```
antlr4 Toto.g4          // produces several Java files
javac *.java            // compiles into xx.class files
grun Toto r             // Run analyzer with starting rule r
```

Lexing with ANTLR: example

Lexing rules:

- Must start with an upper-case letter;
- Follow the usual extended regular-expressions syntax (same as egrep, sed, ...).

A simple example

```
grammar Hello;

// This rule is actually a parsing rule
r   : HELLO ID ; // match "hello" followed by an identifier

HELLO : 'hello' ;          // beware the single quotes
ID : [a-z]+ ;              // match lower-case identifiers
WS : [ \t\r\n]+ -> skip ;  // skip spaces, tabs, newlines
```

Lexing - more than regular languages

Counting in ANTLR - CountLines.g4

```
lexer grammar CountLines;

// Members can be accessed in any rule
@members {int nbLines=0;}

NEWLINE : [\r\n] {
  nbLines++;
  System.out.println("Current lines:"+nbLines);
} ;

SK : ([a-z]+|[ \t]+) -> skip ;
```

```
antlr4 Toto.g4            // produces several Java files
javac *.java              // compiles into xx.class files
grun Toto 'tokens'        // Run the lexical analyser only
```

Outline

What's Parsing ?

Relate tokens by structuring them.

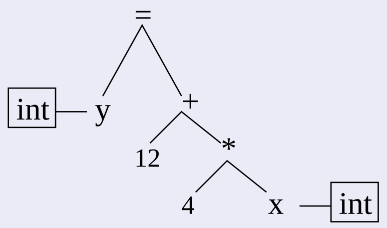
Flat tokens

[TINT, VAR("y"), EQ, INT(12), PLUS, INT(4), FOIS, VAR("x"), PVIRG]

⇒ **Parsing** ⇒

Yes/No +

Structured tokens



Analysis Phases

$$\begin{tikzcd}
\text{source code} \arrow[d] \\
\boxed{\text{lexical analysis}} \arrow[d] \\
\text{sequence of ``lexems'' (\textit{tokens})} \arrow[d] \\
\boxed{\text{syntactic analysis (Parsing)}} \arrow[d] \\
\text{abstract syntax tree (\textit{AST})} \arrow[d] \\
\boxed{\text{semantic analysis}} \arrow[d] \\
\text{abstract syntax (+ symbol table)}
\end{tikzcd}$$

What's behind ?

From a Context-free Grammar, produce a Stack Automaton (see **LIF15**).

Tools: parser generators

- Parser generator: builds a stack automaton from a grammar definition;
- Ex: yacc(C), javacup (Java), OCamlyacc, **ANTLR**, ...
- **input** : a set of grammar rules with actions (`Toto.g4`);
- **output** : a file(s) (`Toto.java`) that contains the corresponding stack automaton.

Lexing vs Parsing

- Lexing supports ($\simeq$ regular) languages;
- We want more (general) languages $\Rightarrow$ rely on context-free grammars;
- To that intent, we need a way:
  - To declare terminal symbols (**tokens**);
  - To write grammars.

▶ Use both Lexing rules and Parsing rules.

From a grammar to a parser

The grammar must be **context-free**:

```
S-> aSb
S-> eps
```

- The grammar rules are specified as **Parsing rules**;
- a and b are terminal tokens, produced by Lexing rules.

On board: notion of derivation tree (see also exercise session2)

Parsing with ANTLR: example 1/2

AnBnLexer.g4

```
lexer grammar AnBnLexer;

// Lexing rules: recognize tokens
A: 'a' ;
B: 'b' ;

WS : [ \t\ r\n ]+ -> skip ; // skip spaces, tabs, newlines
```

Parsing with ANTLR: example 2/2

AnBnParser.g4

```
parser grammar AnBnParser;
options {tokenVocab=AnBnLexer;} // extern tokens definition

// Parsing rules: structure tokens together
prog : s EOF ; // EOF: predefined end-of-file token
s : A s B
    | ;  // nothing for empty alternative
```

ANTLR expressivity

LL(*)

> *At parse-time, decisions gracefully throttle up from conventional fixed $k \geq 1$ lookahead to arbitrary lookahead.*

Further reading (PLDI'11 paper, T. Parr, K. Fisher)

`http://www.antlr.org/papers/LL-star-PLDI11.pdf`

Left recursion

ANTLR permits left recursion:

```
a: a b;
```

But not indirect left recursion.

$$\begin{tikzcd} X_1 \arrow[r] & \dots \arrow[r] & X_n \arrow[ll, bend right] \end{tikzcd}$$

There exist algorithms to eliminate indirect recursions.

Lists

ANTLR permits lists:

```
prog: statement+ ;
```

Read the documentation!

`https://github.com/antlr/antlr4/blob/master/doc/index.md`

Outline

Semantic actions

Semantic actions: code executed each time a grammar rule is matched.

Printing as a semantic action in ANTLR

```
s : A s B { System.out.println("rule s"); }
s : A s B { print("rule s"); }//python
```

Right rule : Python/Java/C++, depending on the back-end

```
antlr4 -Dlanguage=Python2
```

- We can do more than acceptors.

Semantic actions - attributes

An attribute is a set attached to non-terminals/terminals of the grammar

They are usually of two types:

- synthetized: sons $\rightarrow$ father.
- inherited: the converse.

Semantic attributes for numerical expressions 1/2

$$\begin{array}{rcll} e & ::= & c & \textit{constant} \\ & | & x & \textit{variable} \\ & | & e+e & \textit{add} \\ & | & e\times e & \textit{mult} \\ & | & \ldots & \end{array}$$

Let's come to an attribution. On board.

Semantic attributes 2/2 : Implem

Implementation of the former actions (java):

ArithExprParser.g4

```
parser grammar ArithExprParser;
options {tokenVocab=ArithExprLexer;}

prog : expr EOF { System.out.println("Result: "+$expr.val); } ;

expr returns [ int val ] : // expr has an integer attribute
  LPAR e=expr RPAR { $val=$e.val; }
| INT { $val=$INT.int; } // implicit attribute for INT
| e1=expr PLUS e2=expr // name sub-parts
  { $val=$e1.val+$e2.val; } // access attributes
| e1=expr MINUS e2=expr { $val=$e1.val-$e2.val; }
;
```

Writing Evaluators

MIF08

Laure Gonnord
`Laure.Gonnord@univ-lyon1.fr`

Outline

Analysis Phases

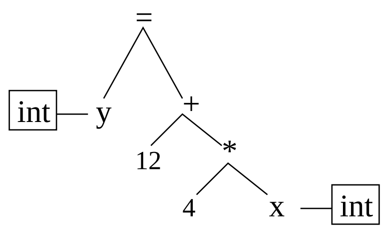
$$\begin{tikzcd}[row sep=small]
\text{source code} \arrow[d] \\
\boxed{\text{lexical analysis}} \arrow[d] \\
\text{sequence of “lexems” (\textit{tokens})} \arrow[d] \\
\boxed{\text{syntactic analysis (Parsing)}} \arrow[d] \\
\text{abstract syntax tree (\textit{AST})} \arrow[d] \\
\boxed{\text{semantic analysis}} \arrow[d] \\
\text{abstract syntax (+ symbol table)}
\end{tikzcd}$$

Until now

We have parsed, and evaluate in semantic actions. But we want:

- more structure.
- an easier way to perform actions (not in the .g4 file).

Notion of Abstract Syntax Tree



- AST: memory representation of a program;
- Node: a language construct;
- Sub-nodes: parameters of the construct;
- Leaves: usually constants or variables.

Separation of concerns

- The semantics of the program could be defined in the semantic actions (of the grammar). Usually though:
  - Syntax analyzer only produces the AST;
  - The rest of the compiler directly **works with this AST**.
- Why ?
  - Manipulating a tree (AST) is easy (recursive style);
  - Separate language syntax from language semantics;
  - During later compiler phases, we can assume that the AST is **syntactically correct** ⇒ simplifies the rest of the compilation.

Running example : Numerical expressions

This is an **abstract syntax** (no more parenthesis, . . .):

$$\begin{array}{rrll} e & ::= & c & \textit{constant} \\ & | & x & \textit{variable} \\ & | & e+e & \textit{add} \\ & | & e\times e & \textit{mult} \\ & | & \dots & \end{array}$$

Let us construct an AST to:

- Evaluate this expression (by tree traversal)
- Later: generate code for these expressions (by tree traversal)

Outline

Outline

Explicit construction of the AST

- Declare a type for the abstract syntax.
- Construct instances of these types during parsing (trees).
- Evaluate with tree traversal.

Example in Java 1/3

AST definition in Java: one class per language construct.

```
public class APlus extends AExpr {
    AExpr e1,e2;

    public APlus (AExpr e1,AExpr e2) { this.e1=e1; this.e2=e2; }

}
public class AMinus extends AExpr { ...
```

Example in Java 2/3

The parser builds an AST instance using AST classes defined previously.

ArithExprASTParser.g4

```
parser grammar ArithExprASTParser ;
options {tokenVocab=ArithExprASTLexer;}

prog returns [ AExpr e ] : expr EOF { $e=$expr.e; } ;

// We create an AExpr instead of computing a value
expr returns [ AExpr e ] :
  LPAR x=expr RPAR { $e=$x.e; }
| INT { $e=new AInt($INT.int); }
| e1=expr PLUS e2=expr { $e=new APlus($e1.e,$e2.e); }
| e1=expr MINUS e2=expr { $e=new AMinus($e1.e,$e2.e); }
;
```

Example in Java 3/3

Evaluation is an eval function per class:

AExpr.java

```java
public abstract class AExpr {
    abstract int eval(); // need to provide semantics
}
```

APlus.java

```java
public class APlus extends AExpr {
    AExpr e1,e2;
    public APlus (AExpr e1,AExpr e2) { this.e1=e1; this.e2=e2; }
    // semantics below
    int eval() { return (e1.eval()+e2.eval()); }
}
```

Outline

Principle - OO programming

The visitor design pattern is a way of separating an algorithm from an object structure on which it operates.[...] In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function.

https://en.wikipedia.org/wiki/Visitor_pattern

Application

Designing evaluators / tree traversal in ANTLR-Python
- The ANTLR compiler generates a Visitor class.
- We override this class to traverse the parsed instance.

Example with ANTLR/Python 1/3

AritParser.g4

```
expr:
        expr mdop expr      #multiplicationExpr
      | expr pmop expr      #additiveExpr
      | atom                          #atomExpr
      ;

atom
      :   INT                 #int
      |   ID                  #id
      |   '(' expr ')'        #parens
```

- compilation with `-Dlanguage=Python2 -visitor`

Example with ANTLR/Python 2/3 -generated file

```
class AritVisitor(ParseTreeVisitor):
...
    # Visit a parse tree produced by AritParser#multiplicationExpr.
    def visitMultiplicationExpr(self, ctx):
        return self.visitChildren(ctx)


    # Visit a parse tree produced by AritParser#atomExpr.
    def visitAtomExpr(self, ctx):
        return self.visitChildren(ctx)

..
```

Example with ANTLR/Python 3/3

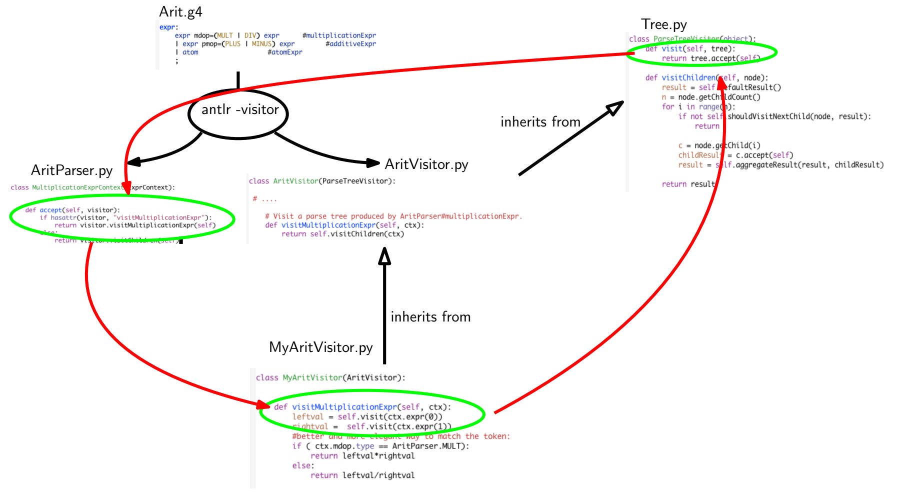
Visitor class overriding to write the evaluator:

MyAritVisitor.py

```
class MyAritVisitor(AritVisitor):
        # Visit a parse tree produced by AritParser#int.
    def visitInt(self, ctx):
        value = int(ctx.getText());
        return value;

    def visitMultiplicationExpr(self, ctx):
        leftval = self.visit(ctx.expr(0))
        rightval =  self.visit(ctx.expr(1))
        myop = self.visit(ctx.mdop())
        if ( myop == '*'):
            return leftval*rightval
        else:
            return leftval/rightval
```

Nice Picture (Lab#3)



```
expr:
    expr mdop=(MULT | DIV) expr      #multiplicationExpr
    | expr pmop=(PLUS | MINUS) expr        #additiveExpr
    | atom                     #atomExpr
    ;
```

```
class ParseTreeVisitor(object):
    def visit(self, tree):
        return tree.accept(self)

    def visitChildren(self, node):
        result = self.defaultResult()
        n = node.getChildCount()
        for i in range(n):
            if not self.shouldVisitNextChild(node, result):
                return

            c = node.getChild(i)
            childResult = c.accept(self)
            result = self.aggregateResult(result, childResult)

        return result
```

```
class MultiplicationExprContext(ExprContext):

    def accept(self, visitor):
        if hasattr(visitor, "visitMultiplicationExpr"):
            return visitor.visitMultiplicationExpr(self)
        else:
            return visitor.visitChildren(self)
```

```
class AritVisitor(ParseTreeVisitor):

 # ....

    # Visit a parse tree produced by AritParser#multiplicationExpr.
    def visitMultiplicationExpr(self, ctx):
        return self.visitChildren(ctx)
```

```
class MyAritVisitor(AritVisitor):


    def visitMultiplicationExpr(self, ctx):
        leftval = self.visit(ctx.expr(0))
        rightval =  self.visit(ctx.expr(1))
        #better and more elegant way to match the token:
        if ( ctx.mdop.type == AritParser.MULT):
            return leftval*rightval
        else:
            return leftval/rightval
```

From grammars to evaluators - summary

- The meaning of each operation/grammar rule is now given by the implementation of the associated function in the visitor.
- The visitor performs a tree traversal on the structure of the parse tree.

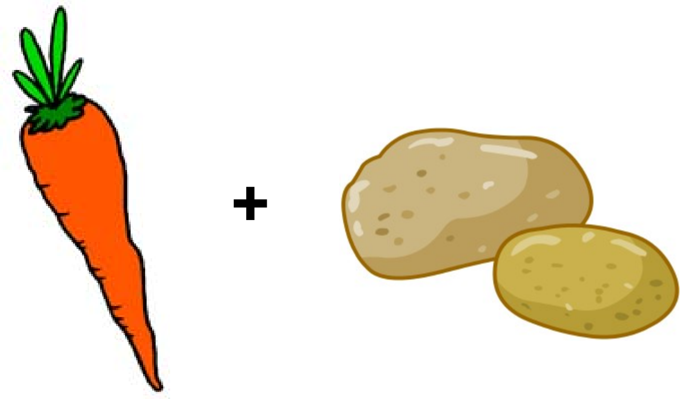
Types, Typing

MIF08

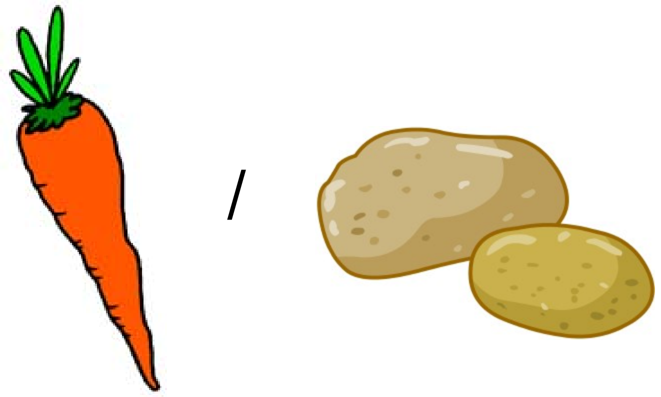
Laure Gonnord
Laure.Gonnord@univ-lyon1.fr

oct 2016

Typing



Typing



Typing

If you write: `"5" + 37`
what do you want to obtain

- a compilation error? (OCaml)
- an exec error? (Python)
- the int 42? (Visual Basic, PHP)
- the string `"537"`? (Java)
- anything else?

and what about `37 / "5"` ?

Typing

When is

```
e1 + e2
```

legal, and what are the semantic actions to perform ?

▶ Typing: an analysis that gives a type to each subexpression, and reject incoherent programs.

When

- Dynamic typing (during exec): Lisp, PHP, Python
- Static typing (at compile time): C, Java, OCaml

▶ Here: the second one.

Slogan

well typed programs do not go wrong

Typing objectives

- Should be **decidable**.
- It should reject programs like `(1 2)` in OCaml, or `1+"toto"` in C before an actual arror in the eveluation of the expression: this is **safety**.
- The type system should be expressive enough and not reject too many programs. (**expressivity**)

Several solutions

- All sub-expressions are anotated by a type

  $\texttt{fun}\ (x:\texttt{int}) \rightarrow \texttt{let}\ (y:\texttt{int}) = (+:)(((x:\texttt{int}),(1:\texttt{int})):\texttt{int}\times\texttt{int})\ \texttt{in}$

  easy to verify, but tedious for the programmer
- Annotate only variable declarations (Pascal, C, Java, . . .)

  $$\texttt{fun}\ (x:\texttt{int}) \rightarrow \texttt{let}\ (y:\texttt{int}) = +(x,1)\ \texttt{in}\ y$$
- Only annotate function parameters

  $$\texttt{fun}\ (x:\texttt{int}) \rightarrow \texttt{let}\ y = +(x,1)\ \texttt{in}\ y$$
- Do nothing : complete inference : Ocaml, Haskell, . . .

Properties

- *correction*: “yes” implies the program is well typed.
- *completeness*: the converse.

(optional)

- *principality* : The most general type is computed.

Outline

Mini-While Syntax

Expressions:

$$\begin{array}{lcll} e & ::= & c & \textit{constant} \\ & | & x & \textit{variable} \\ & | & e+e & \textit{addition} \\ & | & e\times e & \textit{multiplication} \\ & | & \ldots & \end{array}$$

Mini-while:

$$\begin{array}{lcll} S(Smt) & ::= & x := expr & \text{assign} \\ & | & skip & \text{do nothing} \\ & | & S_1; S_2 & \text{sequence} \\ & | & \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 & \text{test} \\ & | & \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done} & \text{loop} \end{array}$$

Typing judgement

We will define how to compute **typing judgements** denoted by:

$$\Gamma \vdash e : \tau$$

and means "in environment Γ, expression e has type τ"

▶ Γ associates a type $\Gamma(x)$ to all free variables x in e.
Here types are basic types: Int|String|Bool

Typing rules for expr

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \texttt{int}}\text{(or bool, ...)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

Hybrid expressions

What if we have `1.2 + 42` ?

- reject?
- compute a float!

▶ This is **type coercion**.

More complex expressions

What if we have types `pointer of bool` or `array of int`? We might want to check equivalence (for addition . . .).

▶ This is called **structural equivalence** (see Dragon Book, "type equivalence"). This is solved by a basic graph traversal.

Typing rules for statements

Idea: the type is void otherwise “typing error”

$$\frac{\Gamma \vdash e : t \quad \Gamma(x) : t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x := e : \texttt{void}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done} : \texttt{void}}$$

Outline

Principle of type checking

- Gamma is constructed with lexing information or parsing (variable declaration with types).
- Rules are semantic actions. The semantic actions are responsible for the evaluation order, as well as typing errors.

Type Checking V1 : visitor

MyMuTypingVisitor.py

```
    def visitAdditiveExpr(self, ctx):
        lvaltype = self.visit(ctx.expr(0))
        rvaltype = self.visit(ctx.expr(1))

        op = self.visit(ctx.oplus())
        if  lvaltype == rvaltype:
            return lvaltype
        elif {lvaltype, rvaltype} == {BaseType.Integer, BaseType
            .Float}:
            return BaseType.Float
        elif op == u'+' and any(vt == BaseType.String for vt in
            (rvaltype, lvaltype)):
            return BaseType.String
        else:
            raise SyntaxError("Invalid type for additive operand
                ")
```

Typing is more than type checking

- Input: Trees are decorated by source code lines.
- Output: Trees are decorated by types.

And we want **informative errors**:

```
Type error at line 42
```

is not sufficient!

Type Checking V2: from AST to decorated ASTs

Idea:

- Generate an AST for the parsed file.
- Decorate with types with a tree traversal.

AST type in Python

Ast.py

```
    def __init__(self):
        super(Expression, self).__init__()

""" Expressions """
class BinOp(Expression):
    def __init__(self, left, right):
        super(Expression, self).__init__()
        self.left = left
        self.right = right

class AddOp(BinOp):
```

AST generation in Python

This AST is generated with the ANTLR visitor from our grammar:

MyAritVisitor.py

```
    def visitAdditiveExpr(self, ctx):
        leftval = self.visit(ctx.expr(0))
        rightval = self.visit(ctx.expr(1))
        if ( self.visit(ctx.pmop()) == '+'):  #see lab for a
            better way to match ops
            return AddOp(left=leftval, right=rightval)
        else:
            return SubOp(left=leftval, right=rightval)
```

Code Generation

MIF08

Laure Gonnord
Laure.Gonnord@univ-lyon1.fr

oct 2016

Lyon 1

Big picture

$$\begin{tikzcd}
\text{source code} \arrow[d] \\
\boxed{\text{lexical+syntactic analysis + typing}} \arrow[d] \\
\text{decorated AST} \arrow[d] \\
\boxed{\text{code production (numerous phases)}} \arrow[d] \\
\text{assembly language}
\end{tikzcd}$$

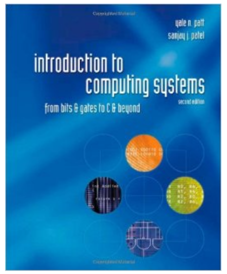
Rules of the Game here

For this code generation:

- Still no functions and no non-basic types. (mini-while)
- Syntax-directed: one grammar rule $\rightarrow$ a set of instructions.
  - ▶ Code redundancy.
- No register reuse: everything will be stored on the stack.

The Target Machine : LC3 (course #1)

[*Introduction to Computing Systems: From Bits and Gates to C and Beyond*, McGraw-Hill, 2004].


yale n. patt
sanjay j. patel
introduction to computing systems
from bits & gates to C & beyond
second edition

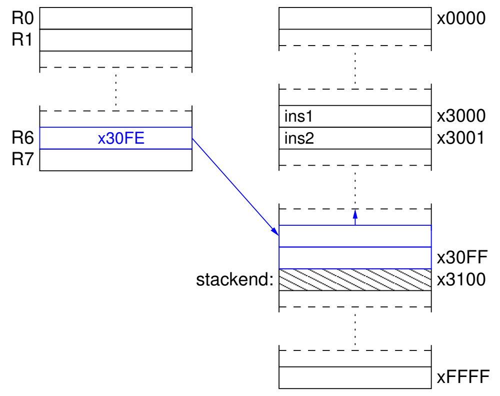
See also:
http://highered.mcgraw-hill.com/sites/0072467509/

A stack, why ?

- Store constants, strings, . . .
- Provide an easy way to communicate arguments values (see later)
- Give place to store intermediate values (here)

LC3 stack emulation - from the archi course

- R6 is initialised to a “end of stack” address (stackend)
- R6 always stores the address of the last value stored in the stack.
- The stack grows in the dir. of **decreasing addresses!**



LC3 stack emulation: concretely 1/2

```
          .ORIG x3000
; Main program
main:     LD R6,spinit   ; stack pointer init
          ...
          HALT

; Stack management
spinit:   .FILL stackend
          .BLKW #15      ; this stack is rather small
stackend: .BLKW #1       ; end of stack address
          .END
```

LC3 stack emulation: concretely 2/2

Push the content of Ri:

```
ADD R6,R6,-1   ; move head of stack
STR Ri,R6,0    ; store the value
```

Pop the content of the stack in Ri:

```
LDR Ri,R6,0    ; pop the value
ADD R6,R6,1    ; head of stack restauration
```

Outline

A first example (1/4)

How do we translate:

```
x=4;
y=12+x;
```

- Compute 4
- Store somewhere `place0`, then link $x \mapsto place0$
- Compute $12 + x$: 12 in `place1`, x in `place2`, then addition, store in `place3`, then link $x \mapsto place3$

▶ the code generator will use a place generator called `newtmp()`

A first example: 3@code (2/4)

"Compute 4 and store in x":

```
AND temp1 temp1 0
ADD temp1 temp1 4
```

And $x \mapsto temp1$.
▶ This is called **three-adress code generation**

A first example: from 3@ code to valid LC-3 (3/4)

But this is not valid LC3 code !
We should use registers, but as they are only 8, we use the stack to store temporaries. Here **store R1 on the stack!**

```
AND R1 R1 0
ADD R1 R1 4
ADD R6 R6 -1 #here also store x -> R6 somewhere
STR R1 R6 0  #now R1 can be recycled
```

A first example: prelude/postlude 4/4

The rest of the code generation:

```
.ORIG X3000
LEA R6 data
[...]
stop: BR stop
data: .BLKW 42
.END
```

▶ This is valid LC-3 code that can be assembled and executed in Pennsim.

Objective of the rest of the course

3-address LC-3 **Code Generation** for the Mini-While language:

- All variables are int/bool.
- All variables are global.
- No functions

with syntax-directed translation. Implementation in Lab.

Code generation utility functions

We will use:

- A new (fresh) temporary can be created with a `newtemp()` function.
- A new fresh label can be created with a `newlabel()` function.

Outline

Abstract Syntax

Expressions:

e	$::=$	c	constant
	$\mid$	x	variable
	$\mid$	$e + e$	addition
	$\mid$	e or e	boolean or
	$\mid$	$e < e$	less than
	$\mid$	$\dots$	

and statements:

$S(Smt)$	$::=$	$x := expr$	assign
	$\mid$	$skip$	do nothing
	$\mid$	$S_1; S_2$	sequence
	$\mid$	`if` b `then` S_1 `else` S_2	test
	$\mid$	`while` b `do` S `done`	loop

Code generation for expressions, example

e ::= c (cte expr)

```
#not valid if c is too big
dr <-newTemp()
code.add(InstructionAND(dr, dr, 0))
code.add(InstructionADD(dr, dr, c))
return dr
```

- this rule gives a way to generate code for any constant.

Code generation for a boolean expression, example

e $::= e_1 < e_2$

```
dr <-newTemp()
t1 <- GenCodeExpr (e1-e2)    #last write in register
(lfalse,lend) <- newLabels()
code.add(InstructionBRzp(lfalse))       #if =0 or >0 jump!
code.add(InstructionAND(dr, dr, 0))
code.add(InstructionADD(dr, dr, 1))     #dr <- true
code.add(InstructionBR(lend))
code.addLabel(lfalse)
code.add(InstructionAND(dr, dr, 0))     #dr <- false
code.addLabel(lend)
return dr
```

- integer value 0 or 1.

Code generation for commands, example

`if` b `then` $S1$ `else` $S2$

```
dr <-GenCodeExpr(b)  #dr is the last written register
lfalse,lendif=newLabels()
code.add(InstructionBRz(lfalse) #if 0 jump to execute
GenCodeSmt(S1)                  #else (execute S1
code.add(InstructionBR(lendif)) #and jump to end)
code.addLabel(lfalse)
GenCodeSmt(S2)
code.addLabel(lendif)
```

Outline

Drawbacks of the former translation

Drawbacks:

- redundancies (constants recomputations, . . .)
- memory intensive loads and stores.

▶ we need a more efficient data structure to reason on: **the control flow graph (CFG)**. (see next course)

Compilation and Program Analysis (#6) : Intermediate Representations: CFG, DAGs (Instruction Selection and Scheduling), SSA

MIF08

Laure Gonnord
Laure.Gonnord@univ-lyon1.fr

oct 2016

Big picture

$$\begin{tikzcd}
\text{source code} \arrow[d] \\
\boxed{\text{lexical+syntactic analysis + typing}} \arrow[d] \\
\text{decorated AST} \arrow[d] \\
\boxed{\text{code production (numerous phases)}} \arrow[d] \\
\text{assembly language}
\end{tikzcd}$$

In context 1/2

In the last course we saw the need for a better data structure to propagate and infer information. We need:

- A data structure that helps us to reason about the flow of the program.
- Which embeds our three address code.

▶ Control-Flow Graph.

In context 2/2

$$\begin{tikzcd}
\text{decorated AST} \arrow[d] \\
\boxed{\text{IR Construction}} \arrow[d] \\
\text{Control-Flow Graph} \arrow[d] \\
\boxed{\text{Clever analyses/code generation}} \arrow[d] \\
\text{assembly language}
\end{tikzcd}$$

Outline

Definitions

Basic Block

Basic block: largest (3-address LC-3) instruction sequence without label. (except at the first instruction) and without jumps and calls.

CFG

It is a directed graph whose vertices are basic blocks, and edge $B_1 \to B_2$ exists if B_2 can follow immediately B_1 in an execution.

- two optimisation levels: local (BB) and global (CFG)

Identifying Basic Blocks (from 3@code)

- The first instruction of a basic block is called a **leader**.
- We can identify leaders via these three properties:
  1. The first instruction in the intermediate code is a leader.
  2. Any instruction that is the target of a conditional or unconditional jump is a leader.
  3. Any instruction that immediately follows a conditional or unconditional jump is a leader.
- Once we have found the leaders, it is straighforward to find the basic blocks: for each leader, its basic block consists of the leader itself, plus all the instructions until the next leader.

Exercise

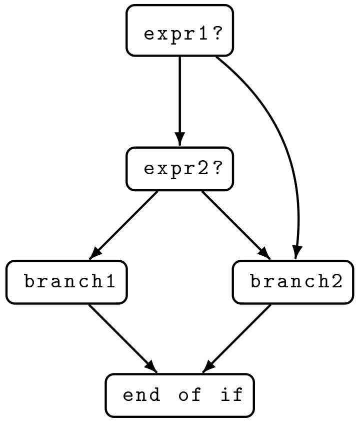
Generate the “high level” CFG for the given program:

```
p:=0;i:=1;
while (i <= 20) do
  if p>60 then
     p:=0;i:=5;
  endif
  i:=2*i+1;
done
k:=p*3;
```

(inside your compiler, blocks will be a list of 3@-LC-3 code)

CFG for tests

```
if (expr1 and expr2)
  ...branch1...
else
  ...branch2...
```



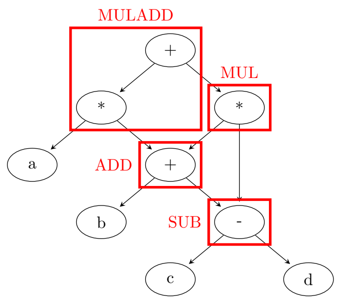
(blocks are subgraphs)

Outline

Big picture

- Front-end → a CFG where nodes are basic blocks.
- Basic blocks → DAGs that explicit common computations

```
u1 := c - d
u2 := b + u1
u3 := a * u2
u4 := u2 * u1
u5 := u3 + u4
```



▶ choose instructions(**selection**) and order them (**scheduling**).

Outline

Instruction Selection

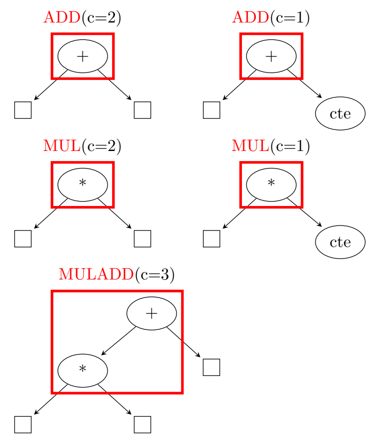
The problem of selecting instructions is a DAG-partitioning problem. But what is the objective ?

The best instructions:

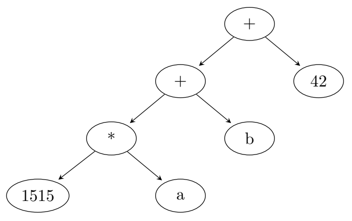
- cover bigger parts of computation.
- cause few memory accesses.

▶ Assign a cost to each instruction, depending on their addressing mode.

Instruction Selection: an example



What is the optimal instruction selection for:



▶ Finding a tiling of minimal cost: it is **NP-complete** (SAT reduction).

Tiling trees / DAGs, in practise

For tiling:

- There is an optimal algorithm for **trees** based on dynamic programing.
- For DAGs we use heuristics (decomposition into a forest of trees, . . .)

▶ The litterature is pletoric on the subject.

Outline

Instruction Scheduling, what for?

We want an evaluation order for the instructions that we choose with **Instruction Scheduling**.

A scheduling is a function θ that associates a **logical date** to each instruction. To be correct, it must respect data dependancies:

```
(S1) u1 := c - d
(S2) u2 := b + u1
```

implies $\theta(S1) < \theta(S_2)$.
▶ How to choose among many correct schedulings? depends on the target architecture.

Architecture-dependant choices

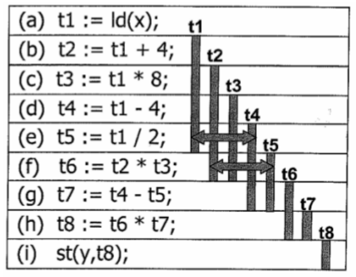
The idea is to exploit the different ressources of the machine at their best:

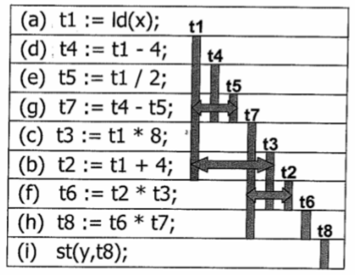
- instruction parallelism: some machine have parallel units (subinstructions of a given instruction).
- prefetch: some machines have non-blocking load/stores, we can run some instructions between a load and its use (hide latency!)
- pipeline.
- registers: see next slide.

(sometimes these criteria are incompatible)

Register use

Some schedules induce less **register pressure**:


```
(a) t1 := ld(x);
(b) t2 := t1 + 4;
(c) t3 := t1 * 8;
(d) t4 := t1 - 4;
(e) t5 := t1 / 2;
(f) t6 := t2 * t3;
(g) t7 := t4 - t5;
(h) t8 := t6 * t7;
(i) st(y,t8);
```


```
(a) t1 := ld(x);
(d) t4 := t1 - 4;
(e) t5 := t1 / 2;
(g) t7 := t4 - t5;
(c) t3 := t1 * 8;
(b) t2 := t1 + 4;
(f) t6 := t2 * t3;
(h) t8 := t6 * t7;
(i) st(y,t8);
```

▶ How to find a schedule with less register pressure?

Scheduling wrt register pressure

Result: this is a linear problem on trees, but NP-complete on DAGs (Sethi, 1975).

▶ Sethi-Ullman algorithm on trees, heuristics on DAGs

Sethi-Ullman algorithm on trees

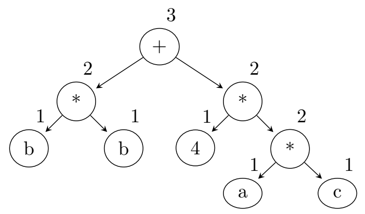
$\rho(node)$ denoting the number of (pseudo)-registers necessary to compute a node:

- $\rho(leaf) = 1$
- $\rho(nodeop(e_1, e_2)) = \begin{cases} max\{\rho(e_1), \rho e_2\} & \text{if } \rho(e_1) \neq \rho(e_2) \\ \rho(e_1) + 1 & \text{else} \end{cases}$

(the idea for non “balanced” subtrees is to execute the one with the biggest ρ first, then the other branch, then the op. If the tree is balanced, then we need an extra register)

▶ then the code is produced with postfix tree traversal, the biggest register consumers first.

Sethi-Ullman algorithm on trees - an example



	tmp_1	tmp_2	tmp_3	tmp_4
`mul tmp1, b b`				
`mul tmp2, a c`	■			
`ldi tmp3, 4`	■	■		
`mul tmp4, tmp2, tmp3`	■	■	■	
`mul tmp5, tmp1 ,temp4`	■			■

Another example

Consider the expression $((a + b) * (a - b) * (a - b)) + 1$ where a and b are stored in **stack slots**. The multiplication will be implemented with the new instruction `mul t1 t2 t3`.

What is the minimum amount of registers required to evaluate E ? Generate code and draw the liveness intervals for your code.

Conclusion (instruction selection/scheduling)

Plenty of other algorithms in the literature:

- Scheduling DAGs with heuristics, . . .
- Scheduling loops (M2 course on advanced compilation)

Practical session:

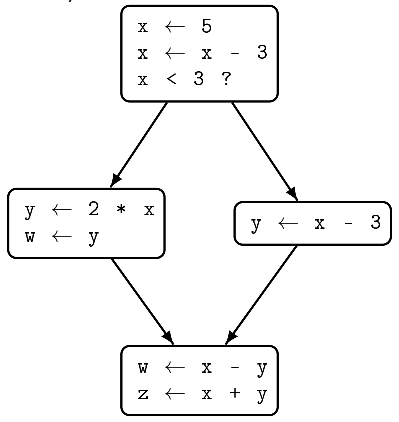
- we have (nearly) no choice for the instructions in the LC3 ISA.
- evaluating the impact of scheduling is a bit hard.

We won’t implement any of the previous algorithms.

Outline

What's SSA? (Cytron 1991)

Each variable is assigned only once (Static Single Assigment form):

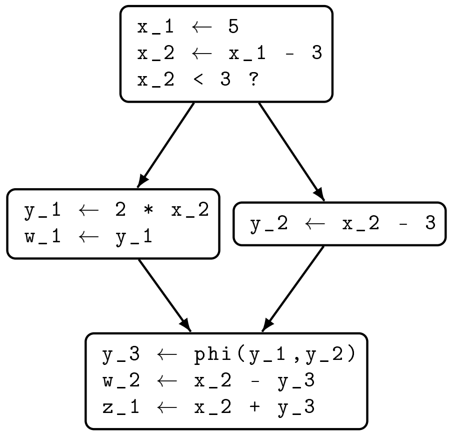


```
x ← 5
x ← x - 3
x < 3 ?
```

```
y ← 2 * x
w ← y
```

```
y ← x - 3
```

```
w ← x - y
z ← x + y
```



```
x_1 ← 5
x_2 ← x_1 - 3
x_2 < 3 ?
```

```
y_1 ← 2 * x_2
w_1 ← y_1
```

```
y_2 ← x_2 - 3
```

```
y_3 ← phi(y_1,y_2)
w_2 ← x_2 - y_3
z_1 ← x_2 + y_3
```

SSA-Graph Construction

See `http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/StaticSingleAssignment.pdf`

Pro/cons

- Another IR, and cost of contruction/deconstruction
+ (some) Analyses/optimisations are easier to perform (like register allocation): `http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/SSABasedRA.pdf`

Compilation and Program Analysis(#7): Register Allocation + Data Flow Analyses

MIF08

Laure Gonnord
`Laure.Gonnord@univ-lyon1.fr`

oct 2016

Where are we ?

$$\begin{tikzcd}[row sep=small]
\text{source code} \arrow[d] \\
\boxed{\text{lexical+syntactic analysis + typing}} \arrow[d] \\
\text{decorated AST} \arrow[d] \\
\boxed{\text{code production (numerous phases)}} \arrow[d] \\
\text{assembly language}
\end{tikzcd}$$

- We work on IRs (Middle-end).

Outline

Credits

Fernando Pereira's course on register allocation:

`http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/RegisterAllocation.pdf`

What for ?

- Finding storage locations to the values manipulated by the program ▶ registers or memory.
- registers are fast but in small quantity.
- memory is plenty, but slower access time.

▶ A good register allocator should strive to keep in registers the variables used more often.

"Because of the central role that register allocation plays, both in speeding up the code and in making other optimizations useful, it is one of the most important - if not the most important - of the optimizations."

Hennessy and Patterson (2006) - [Appendix B; p. 26]

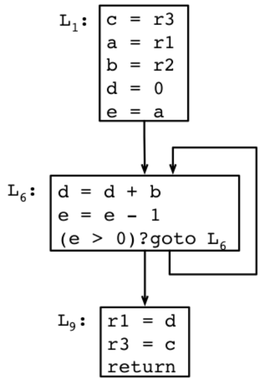
What for ?

Expected behavior of **register allocation**:

- Input: a CFG with basic blocks with 3-address code (and pseudo-registers, aka temporaries)
- Output : same CFG but without pseudo-registers:
  - replace with physical registers as much as possible.
  - if not **splill**, ie allocate a place in memory.
  - all copies assigned to the same physical registers (“moves”) can be removed: **coalescing**

Register constraints

Some variable are assigned to some specific registers (compiler, architecture constraints)



```
L1: c = r3
    a = r1
    b = r2
    d = 0
    e = a

L6: d = d + b
    e = e - 1
    (e > 0)?goto L6

L9: r1 = d
    r3 = c
    return
```

▶ r1,r2,r3 are used to pass function arguments here.

The key notion: liveness

Observation

Two variables that are simultaneously **alive** must be assigned different registers.

(formal definition of alive follows)

Register assignment is NP-complete

Theorem

Given P and K general purpose registers, is there an assignment of the variables P in registers, such that (i) every variable gets at least one register along its entire live range, and (ii) simultaneously live variables are given different registers ?

Gregory Chaitin has shown, in the early 80's, that the register assignment problem is NP-Complete (register allocation via coloring, 1981)

3-phase algorithm

- **Liveness analysis**
  - When is a given value necessary for the rest of the computation?
- **Interference graph**
  - A graph that encodes which pseudo-registers cannot be mapped to the same location.
- **Graph coloring** then register allocation.
  - The effective allocation: physical registers and stack allocation for pseudo-registers.

Outline

Outline

Liveness analysis

In the sequel we call **variable** a pseudo-register or a physical register.

Alive Variable

In a given program point, a variable is said to be *alive* if the value she contains may be used in the rest of the execution.

May: non decidable property ▶ overapproximation.

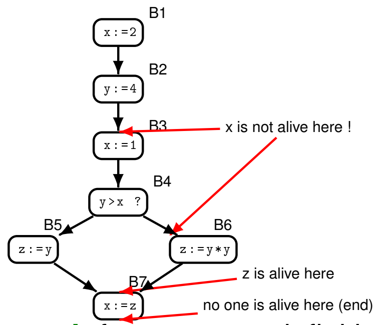
Important remark: here a block = a statement/program point. We have the same kind of analyses with block=basic block.

An example for live ranges

Definition

A variable is **live** at the exit of a block if there exists a path from the block to a use of the variable that does not redefine the variable.

```
x:=2;
y:=4;
x:=1;
if (y>x)
   then z:=y
   else z=y*y ;
x:=z;
```



▶ The information flow is **backward**: from uses to definitions.

Data flow expressions

Definition

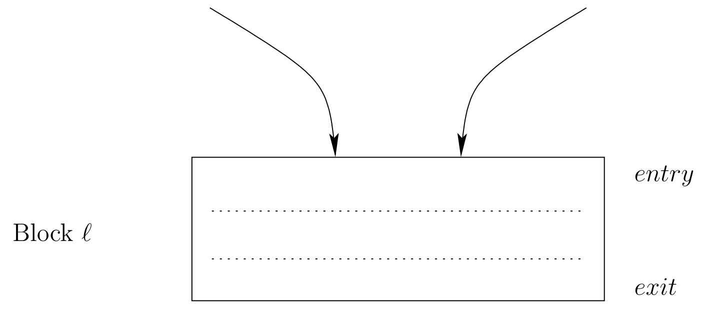
A variable that appears on the left hand side of an assignment is **killed** by the block. Tests do no kill variables.

Definition

A **generated** variable is a variable that appears in the block.

▶ Sets : $kill_{LV}(block)$ and $gen_{LV}(block)$

Data flow expressions



$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell = final \\ \bigcup\{LV_{entry}(\ell') \mid (\ell,\ell') \in flow(G)\} \end{cases}$$

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\right) \cup gen_{LV}(\ell)$$

Data flow equation: solving

Here:

- Initialise LV sets to $\emptyset$.
- Compute LV_{entry} sets, then LV_{exit}, and continue.
- Stop when a fix point is reached.

▶ (vector of) Sets are strictly growing, and the live range set is at most the set of all variables, thus **this algorithm terminates**.

Steps

$LV_{entry}(\ell)$ denoted by $In(\ell)$, $LV_{entry}(\ell)$ by $Out(\ell)$ initilisation to emptysets is not depicted.

			Step 1		Step 2		Step 3 (stable)
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$
1	$\{x\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\{y\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{y\}$	$\emptyset$
3	$\{x\}$	$\emptyset$	$\emptyset$	$\{x,y\}$	$\{y\}$	$\{x,y\}$	$\{y\}$
4	$\emptyset$	$\{x,y\}$	$\{x,y\}$	$\{y\}$	$\{x,y\}$	$\{y\}$	$\{x,y\}$
5	$\{z\}$	$\{y\}$	$\{y\}$	$\{z\}$	$\{y\}$	$\{z\}$	$\{y\}$
6	$\{z\}$	$\{y\}$	$\{y\}$	$\{z\}$	$\{y\}$	$\{z\}$	$\{y\}$
7	$\{x\}$	$\{z\}$	$\{z\}$	$\emptyset$	$\{z\}$	$\emptyset$	$\{z\}$

Final result and use

Backward analysis and we want the smallest sets, here is the final result : (we assume all vars are dead at the end).

ℓ	$LV_{entry}(\ell)$	$LV_{exit}(\ell)$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{y\}$
3	$\{y\}$	$\{x,y\}$
4	$\{x,y\}$	$\{y\}$
5	$\{y\}$	$\{z\}$
6	$\{y\}$	$\{z\}$
7	$\{z\}$	$\emptyset$

▶ Use : Dead code elimination ! Note : can be improved by computing the use-defs paths. (see Nielson/Nielson/Hankin)

Outline

Common subexpressions

Avoiding the computation of an (arithmetic) expression :

```
x:=a+b;
y:=a*b;
while(y>a+b) do
   a:=a+a;
   x:=a+b;
done
```

▶ Same kind of equations.

Outline

Interference

The liveness analysis gives us for $a + (b + c)$:

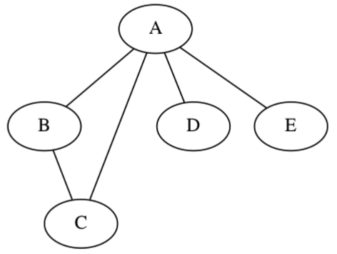
	tmp_1	tmp_2	tmp_3	tmp_4	tmp_5	tmp_6
`ld tmp1,la`						
`ld tmp2,lb`	■					
`ld tmp3,lc`	■	■				
`ADD tmp4, tmp2, tmp3`	■	■	■			
`ADD tmp5, tmp4,0`	■			■		
`ADD tmp6, tmp1, tmp5`	■				■	
⋮						■

▶ `tmp1` is in conflit with `tmp2` (because of instruction 3) denoted by $tmp_1 \bowtie tmp_2$.

Important remark: technically, `ADD tmp5, tmp4,0` is a **move instruction**

Interference graph

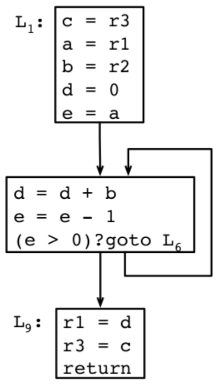
A denotes `tmp1`, . . . $\bowtie$ defines a graph:



We want a **correct allocation** with respect to $\bowtie$:
$tmp_1 \bowtie tmp_2 \implies Alloc(tmp_1) \neq Alloc(tmp_2)$.

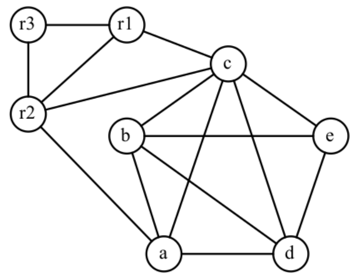
▶ Graph coloring.

Running example


```
L1: c = r3
    a = r1
    b = r2
    d = 0
    e = a

L6: d = d + b
    e = e - 1
    (e > 0)?goto L6

L9: r1 = d
    r3 = c
    return
```



Kempe's simplification algorithm 1/2

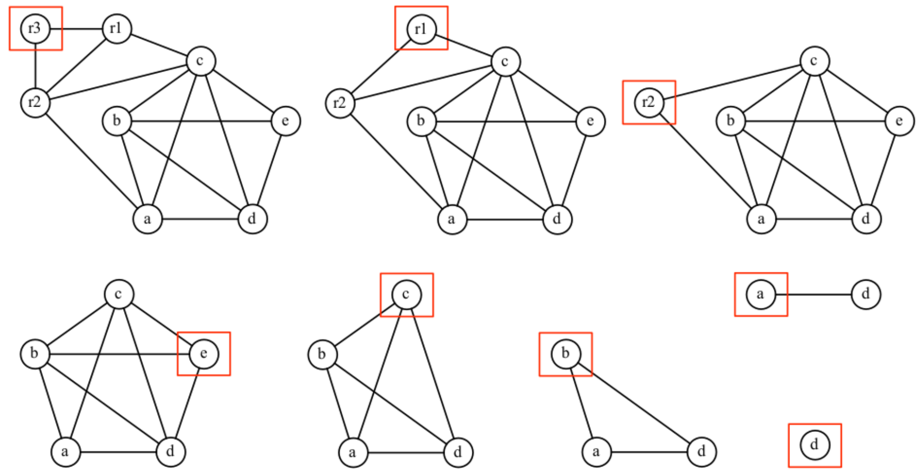
On the interference graph (without coalesce edges):

Proposition (Kempe 1879)

Suppose the graph contains a node m with fewer than K neighbours. Then if $G' = G \setminus \{m\}$ can be colored, then G can be colored as well.

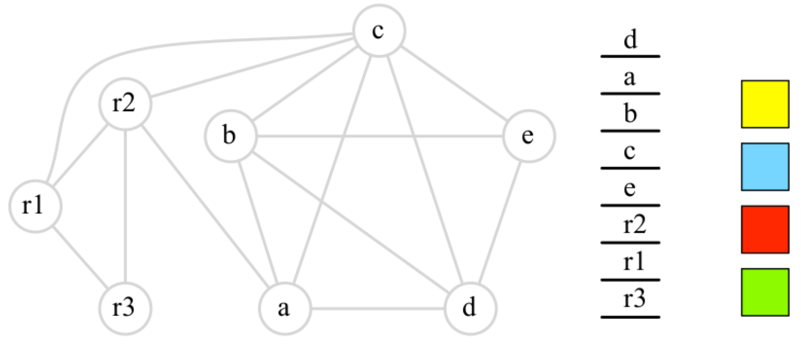
- Pick a low degree node, and remove it, and continue until remove all (the graph is K-colorable) or . . .

Kempe's simplification algorithm 2/2

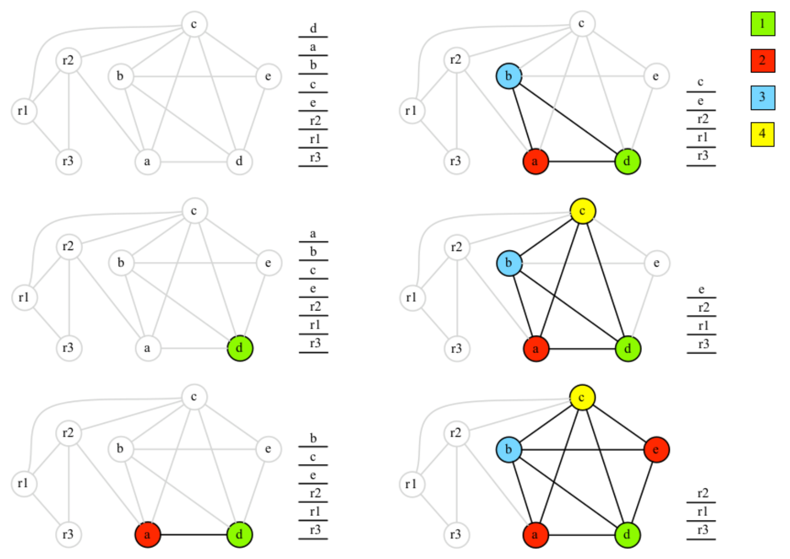


Let's color!

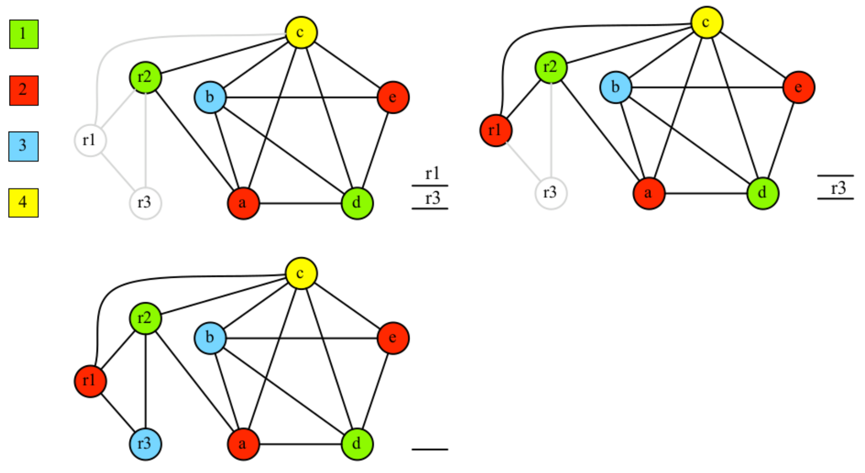
- We assign colors to the nodes greedily, in the reverse order in which nodes are removed from the graph.
- The color of the next node is the first color that is available, *i.e.* not used by any neighbour.



Greedy coloring example 1/2



Greedy coloring example 2/2



If the graph is not colorable

Non-colored variables are named **spilled pseudo-registers**.

Idea: Modify the code to lower the number of simultaneously alive registers. Plenty of solutions, the simplier is to reserve a *dedicated place for a given spilled variable*, and store and load from memory:

```
ADD temp5, temp4, temp3
...
ADD temp6, temp5, #5
```

becomes:

```
ADDINMEMORY [placefortemp5], temp4, temp3
...
ADDxx       temp6, [placefortemp5], #5
```

But we do not have this kind of instruction in our machine!

One solution for spilled variables

We invent 2 versions of the same variable (**live-range splitting**), and modify the code into:

```
ADD temp51, temp4, temp3
ST temp51 [placeinmemory]
..
LD temp52 [placeinmemory]
ADD temp6, temp52, #5
```

- But now we have to allocate these two new variables!

We relaunch the coloring algorithm. This is called iterative register coloring. (see Exercise Sheet 4)

An example

Consider the following assembly code, where t1, . . . , t8 are temporaries to be allocated:

```
ld  t1,[a1]
ld  t2,[a2]
sub t3,t1,t2
ld  t4,[b1]
ld  t5,[b2]
sub t6,t4,t5
MOV(t7,t6)
add t8,t3,t7
```

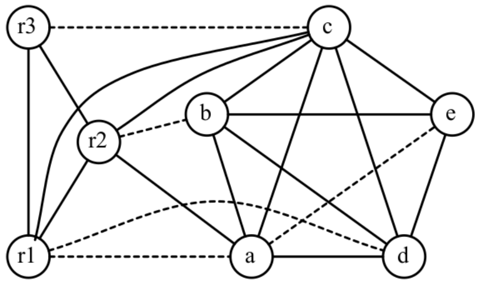
- Draw the liveness intervals and the interference graph.
- Apply the simplification coloring with $K = 3$ registers. Give the final code.
- Apply the **iterative** coloring with $K = 2$ registers. Give the final code.

Physical Memory Allocation

We will invent physical memory places from the stack pointer (see next course).

Other Algorithms

- **Linear scan**: greedy coloring of interval graphs. (see Fernando Pereira's slides on register allocation: 18 to 35)
- **Iterative Register Coalescing** (George/Appel, TOPLAS, 1996) (same, from slides 44), which uses “coalesce edges” (variables are related by move instructions).
- Plenty of other heuristics for splilling.



A nice result

Chordal graphs are P-colorable

For certain classes of graphs, graph coloring is P. This is the case for **cordal graphs** where every cycle with 4 or more edges has a chord (connects 2 vertices in the cycle but not part of the cycle).

Important result (Sebastian Hack): Programs in strict SSA form have this property.

▶ Pereira Palsberg Register allocation (APLAS 2005).

Compilation (#8) : Functions: syntax and code generation

MIF08

Laure Gonnord
Laure.Gonnord@univ-lyon1.fr

nov 2016

Big picture

So far:

- All variables were global.
- No function call.

Inspiration: N. Louvet, Lyon1 (archi part), C. Alias (code gen part).

Outline

Concrete syntax 1/2

- we add variable declaration (with the var keyword):

```
vardecl
 : VAR ID ASSIGN expr
 ;
```

- blocks are like before:

```
block
 : stat*    #statList
 ;
stat_block
 : OBRACE block CBRACE
 | stat
;
```

- procedures declaration:

```
declproc:
 : PROC ID IS stat
;
```

Concrete syntax 2/2

And now there will two new kinds of statements:

```
stat
 : assignment
 | if_stat
 | while_stat
 | log
 | CALL ID
 | BEGIN declvar* declproc* block_stat END
 ;
```

- We can declare local procedures inside local procedures.

On board : add new concrete syntax for **functions**.

Abstract syntax

WLOG, we will only consider programs with procedures:

$$\begin{array}{lll}
S & \in & \textbf{Stm} \\
S & ::= & x := a \mid \text{skip} \mid S_1; S_2 \mid \\
 & & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
 & & \text{while } b \text{ do } S \text{ od } \mid \text{begin } D_V\ D_P; S \text{ end} \mid \text{call } p \\
D_V & ::= & \text{var } x := a;\ D_V \mid \epsilon \\
D_P & ::= & \text{proc } p \text{ is } S;\ D_P \mid \epsilon
\end{array}$$

Exercise

EX : syntax for functions

Outline

A bit about Typing

Two important remarks:

- Now that variables are local, the typing environnement should also be updated each time we enter a procedure.
- Type checking for functions: construct the type from definitions, check when a call is performed (see the course on typing ML).

Outline

Routines

A procedure/routine in assembly is just a piece of code

- its first instruction's address is known and tagged with a label.
- the JSR instruction jumps to this piece of code (routine call).
- at the end of the routine, a RET instruction is executed for the PC to get the address of the instruction after the routine call.

Slides coming from the architecture course, N. Louvet

Routines in LC-3, how? JSR

When a routine is called, we have to store the address where to come back:

- syntax : JSR label
- action : R7 <- PC ; PC <- PC + SEXT(PCoffset11)
  - -1024≤Sext(Offset11)≤1023.
  - if adI is the JSR instruction's address, the branching address is:

    adM = adI+1+Sext(PCOffset11), with
    adI $- 1023 \leq$ adM $\leq$ adI $+ 1024$.

Routines in LC-3, how RET

Inside the routine code, the RET instruction enables to come back:

- syntax : RET
- action : PC <- R7

Writing routines

Call to the `sub` routine:

```
 ...
JSR sub    ; R7 <- next line address
...
```

The last instruction of the routine is RET :

```
; sub routine
sub:  ...
      ...
      RET        ; back to main
```

An example - strlen, without routine

```
        .ORIG x3000
        LEA R0,string   ;
        AND R1,R1,0     ;
loop:   LDR R2,R0,0     ;
        BRz end         ;
        ADD R0,R0,1     ;
        ADD R1,R1,1     ;
        BR loop
end:    ST R1,res
        HALT
; Constant chain
string: .STRINGZ "Hello World"
res:    .BLKW #1
        .END
```

String length routine 1/2

`strlen` call (the result will be stored in R0).

```
       .ORIG x3000
; Main program
        LEA R0,string  ; R0 <- @(string)
        JSR strlen      ; routine call
        ST R0,lg1
        HALT

; Data
string: .STRINGZ "Hello World"
lg1:    .BLKW #1
```

String length routine 2/2

```
strlen: AND R1,R1,0     ;
loop:   LDR R2,R0,0     ;
        BRz end         ;
        ADD R0,R0,1     ;
        ADD R1,R1,1     ;
        BR loop
end:    ADD R0,R1,0     ; R0 <- R1
        RET             ; back to main (JMP R7)

        .END            ; END of complete prog
```

Routines in LC-3: chaining routines

If a routine needs to **call another one**:

- Some temporary registers may have to be stored somewhere
- Its return address (in R7!) needs also to be stored.

▶ Store in the stack (R6) before, restore after.

Outline

Rules of the game

We still have our LC3 machine with registers:

- general purpose registers R0 to R5.
- a stack pointer (SP), here R6.
- a frame pointer (FP), here R7

Simplification: **no imbricated** function declaration.

▶ when `call p`, there is a unique `p` code labeled by p :

Key notion: activation record - Vocabulary 1/2

(picture needed)

- Any execution instance of a function is called an **activation**.
- We can represent all the activations of a given program with an **activation tree**.

Key notion: activation record - Vocabulary 2/2

During execution, we need to keep track of alive activations:

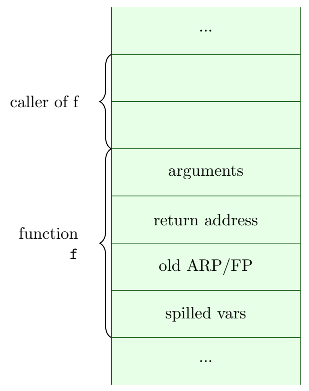
- Control stack
- An activation is pushed when activated
- When its over, it is poped out.

▶ **Notion of activation record** that stores the information of one function call at execution.

▶ **The compiler** is in charge of their management.

Slides inspired by C. Alias

Activation record of a given function



The frame pointer (ARP or FP) points to the current activation record (first spilled variable).

Code generation 1/2

For functions, we have to reserve (local) place before knowing the number of spilled variables!

int f(x1,x2) S; return e

```
code.addMacro(PUSH R7)        #store @ret
code.addcopy(R6,R7)           #R7<-R6
code.addCode(ADD R6 R6 xx)    #xx= future nb of spilled vars
code.addCode(LDR tmp1 R7 -1)  #arg1
code.addCode(LDR tmp2 R7 -2)  #arg2 (in rev order)
CodeGenSmt(S)                 #under the context x1->tmp1...
dr<-CodeGen(e)                #same!
code.addcopy(dr,R0)  #convention return val in R0
code.addMacro(RET,2+xx) #desalloc args + spilled vars + retur
```

▶ `CodeGenSmt` must be called with a modified map.

Code generation 2/2

call f(e1,e2)	

	`Gencodesaveregisters()   #save current values of reg.`
	`dr <- newtmp`
	`dr1=Gencode(e1)`
	`code.addMacro(PUSH dr1)`
	`dr2=Gencode(e2)`
	`code.addMacro(PUSH dr2)`
	`code.add(JSR f)     #return @ in R7`
	`code.addcopy(r0,dr) # dr <- returned value`
	`Gencoderestoreregisters() #restore curr values of reg.`
	`return dr`

A simple example 1/3

Generate code and draw the activation records during the call execution of f:

```
int f(x) {return x+1;}

main:
z:=f(7);
```

A simple example 2/3

```
main:
PUSH(R0,R1....R5)   #should be replaced by R6 manipulation.
AND tmp1 tmp1 0
ADD tmp1 temp1 7
PUSH(tmp1)
JSR f
AND tmp2 tmp2 0
AND tmp2 R0 0
pop(R5... ,R1,R0) #but not the register associated to temp2
[use of temp2 here]
```

A simple example 3/3

```
f: PUSH(R7)
   COPY(R6,R7)
   ADD R6 R6 xx       #xx=number of spilled vars
   LDR tmp1 R7 #1     #first argument
   ADD tmp2 tmp1 1
   COPY(tmp2,R0)      #store result in R0
   COPY(R7,R6)        #this is postlude
   ADD R6 R6 -1      #1 argument
   POP(R7)
```

Register allocation gives $tmp1, tmp2$ are allocated in $R1$ (or R0 if we are clever). Thus `xx`=0.

To go further

- How to implement the different calling conventions? (here, call by value)?
- How to implement imbricated functions (dynamic link, static link).
- How to store more complex types (arrays, structs, user defined types)?