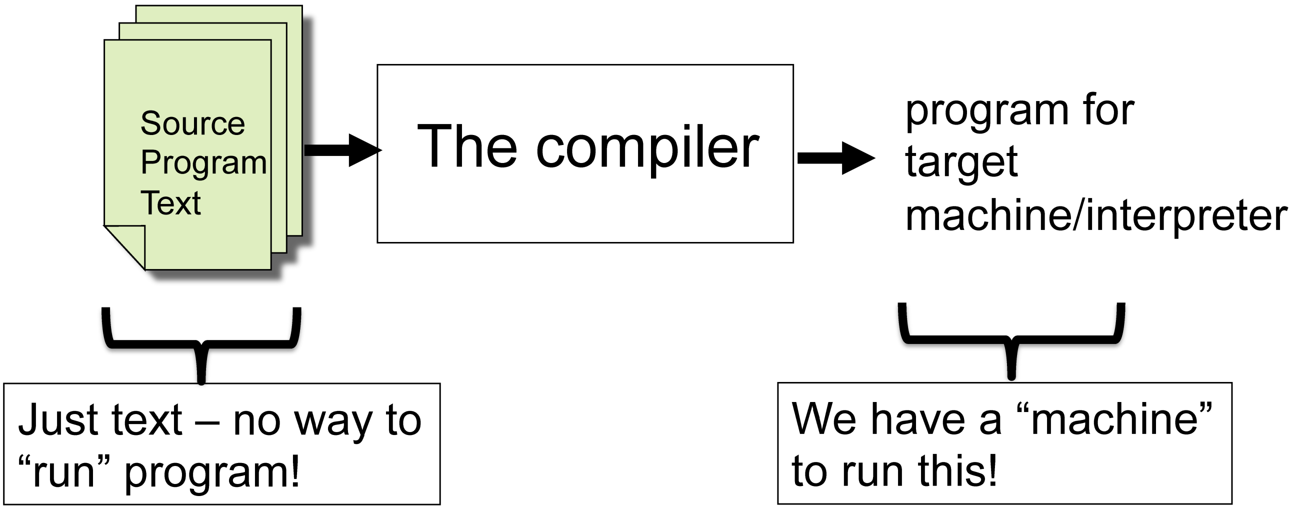
Compiler Construction
Lent Term 2013

Lectures 1 - 4 (of 16)

Timothy G. Griffin
tgg22@cam.ac.uk

Computer Laboratory
University of Cambridge

Compilation is a special kind of translation



A good compiler should …

This course!
- **be correct in the sense that meaning is preserved**
- **use good low-level representations**
- **produce usable error messages**

OptComp, Part II
- **generate efficient code**

General software engineering
- **be efficient**
- **be well-structured and maintainable**

Pick any 2? (generate efficient code / be efficient / be well-structured and maintainable)

Why Study Compilers?

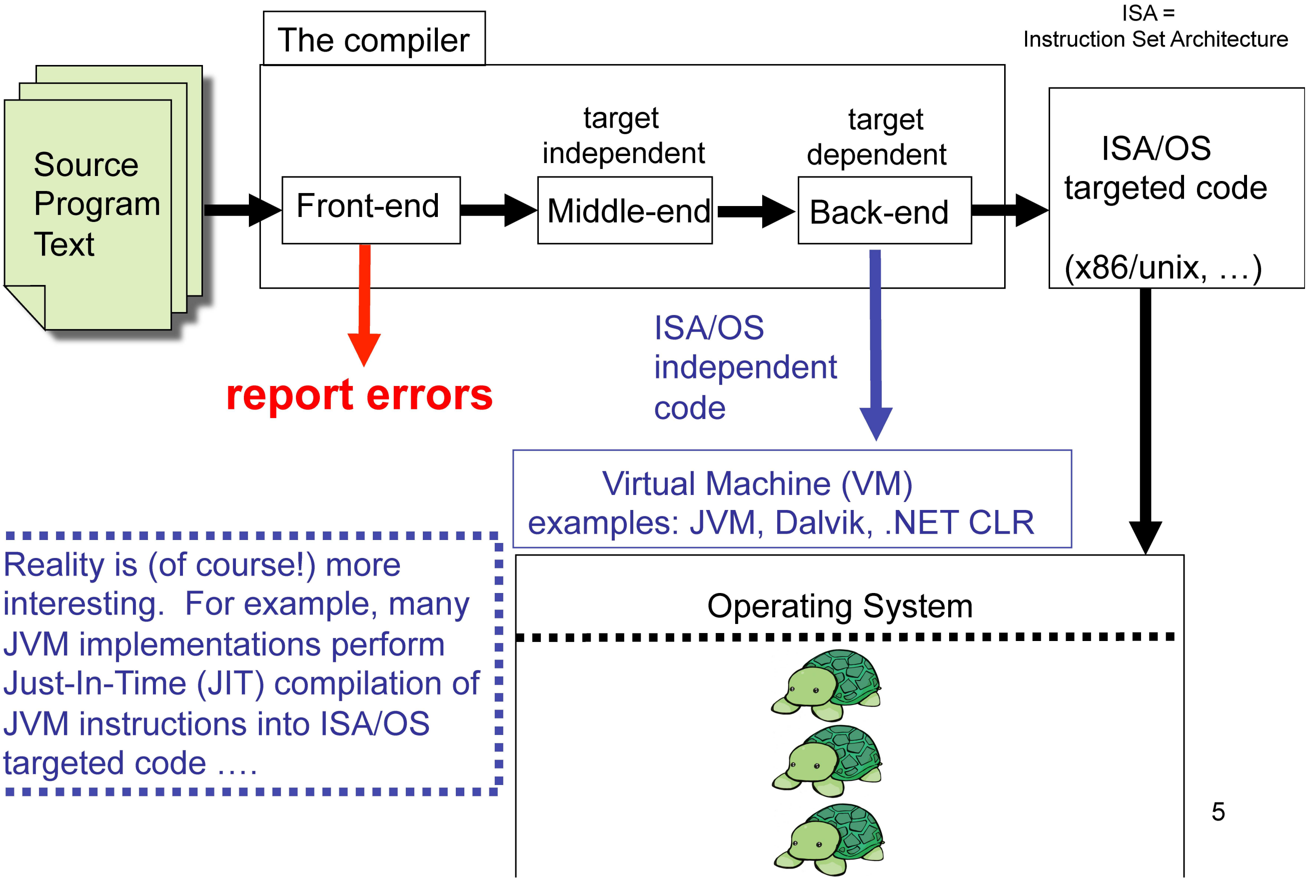
- **Although many of the basic ideas were developed over 40 years ago, compiler construction is still an evolving and active area of research and development.**
- **Compilers are intimately related to programming language design and evolution. Languages continue to evolve.**
- **Renewed demand for compiler skills in industry (mostly due to mobile devices?)**
- **Every Computer Scientist should have a basic understanding of how compilers work.**

Mind The Gap

High Level Language	Typical Target Language
• Machine independent	• Machine specific
• Complex syntax	• Simple syntax
• Complex type system	• Simple types
• Variables	• memory, registers, words
• Nested scope	• Single flat scope
• Procedures, functions	
• Objects	
• Modules	
• …	

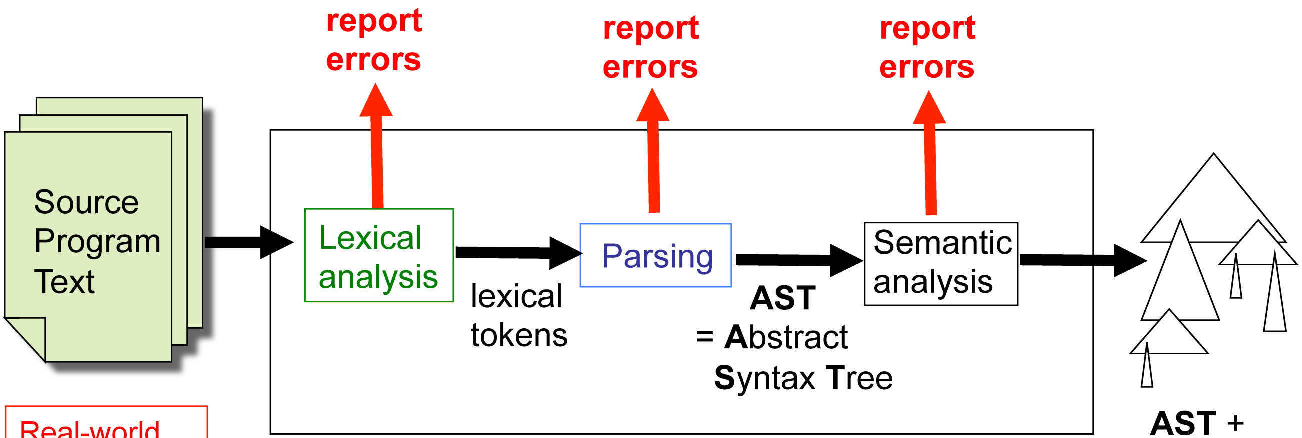
Help!!! Where do we begin???

Conceptual view of a typical compiler



Reality is (of course!) more interesting. For example, many JVM implementations perform Just-In-Time (JIT) compilation of JVM instructions into ISA/OS targeted code ….

The shape of a typical "front-end"



Real-world compilers may collapse some or all of these stages

Lexical theory based on finite automaton and regular expressions

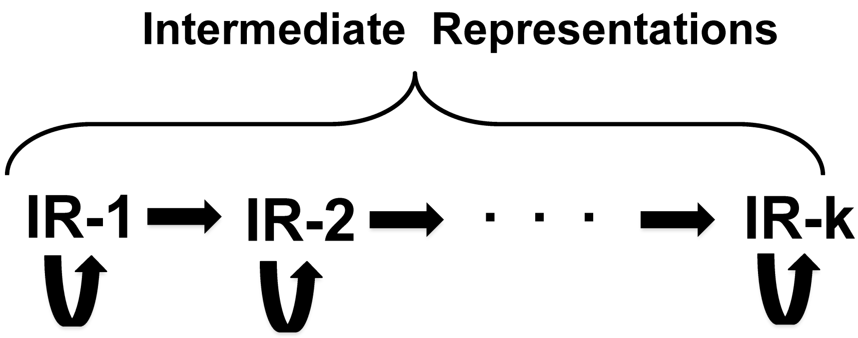
Parsing Theory based on push-down automaton and context-free grammars

Enforce "static sematics" of language: type checking, def/use rules, and so on (**SPL**!)

The AST output from the front-end should represent a <u>legal program</u> in the source language. ("Legal" of course does not mean "bug-free"!)

SPL = Semantics of Programming Languages, Part 1B

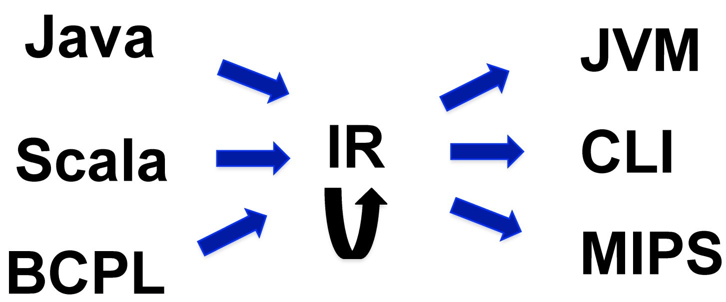
Our view of the middle- and back-ends : a sequence of small transformations



Of course industrial-strength compilers may collapse many small-steps …

- Each **IR** has its own semantics (perhaps informal)
- Each transformation (➡) preserves semantics (**SPL**!)
- Each transformation eliminates only a few aspects of **the gap** (so **IR**-(i+1) is at a "lower level" than **IR**-i)
- Each transformation is fairly easy to understand
- Some transformations can be described as "optimizations"
- In principle (but not in practice), each **IR** could be associated with its own formal semantics and machine/interpreter

Another view (often seen in textbooks)



- One **IR** to rule them all
- Difficult to derive an **IR** if one has never seen a compiler before
- For instructional purposes we prefer to introduce multiple **IR**s

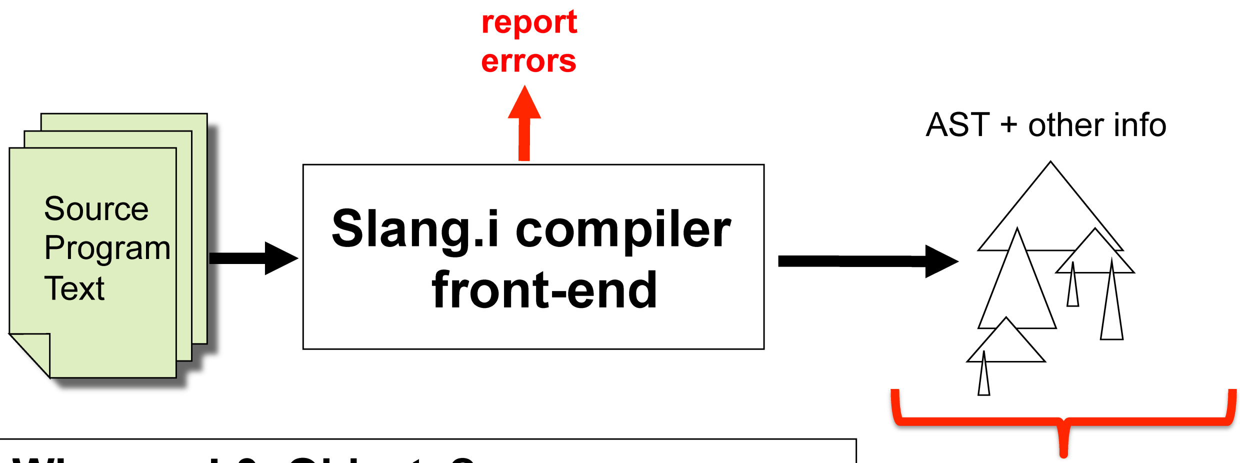
Simple language (Slang) compilers

The lectures will center around *compiler concepts,* mostly illustrated by developing Slang compilers.

We start with **Slang.1**, a very simple simple language and progress to more complex **Slang.2**, **Slang.3**, **Slang.4**:

- **Slang.1** : simple imperative language with only assignment, if-then-else, and while loops
- **Slang.2** : extend language with scope structure, simple functions/procedures
- **Slang.3** : extend language with tuples, records, and first-order functions
- **Slang.4** : extend language with objects

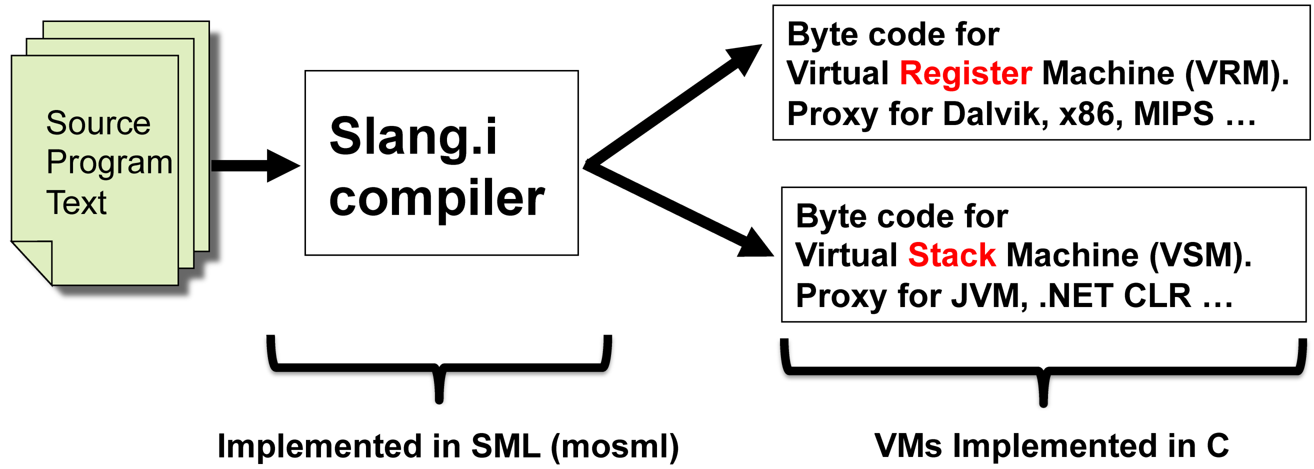
Slang is (bad?) concrete syntax for SPL languages



- **Why use L3+Objects?**
- **Why define yet another toy language?**
- **SPL gives us <u>clear type system</u>**
- **SPL gives us <u>clear semantics</u>**
- **L3+Objects covers most of the features we want to talk about!**

This will always be in some subset of "L3+Objects" from **Semantics of Programming Languages (SPL)**

Slang compiler targets two machines



- Prototype implementations available on course website
- Tripos will be about concepts, not details of this code.
- I have avoided advanced features of SML and C
- Programs written for clarity, not efficiency
- Bug reports appreciated, but only with a fix proposed!

The Shape of this Course

illustrated with	Lecture	Concepts
Slang.1 VRM.0 and VSM.0	1.	Overview
	2.	Simple lexical analysis, recursive descent parsing (thus "bad" syntax), and simple type checking
	3.	Targeting a Virtual Register Machine (VRM)
	4.	Targeting a Virtual Stack Machine (VSM) . Simple "peep hole" optimization
Slang.2 VRM.1 and VSM.1 (call stack extensions)	5.	Block structure, simple functions, stack frames
	6.	Targeting a VRM, targeting a VSM
Slang.3 VRM.2 and VSM.2 (heap and instruction set extensions)	7.	Tuples, records, first-class functions. Heap allocation
	8.	More on first-class functions and closures
	9.	Improving the generated code. Enhanced VM instruction sets, improved instruction selection, more "peep hole" optimization, simple register allocation for VRM
	10.	Memory Management ("garbage collection")
	11.	Assorted topics : Bootstrapping, Exceptions
Slang.4 VRM.2 and VSM.2	12.	Objects (delayed to ensure coverage in SPL), plus linking and loading
mosmllex and mosmlyacc	13.	Return to lexical analysis : application of Theory of Regular Languages and Finite Automata
	14.	Generating Recursive descent parsers
	15.	Beyond Recursive Descent Parsing I
	16.	Beyond Recursive Descent Parsing II

Reading

Printed notes (from previous years --- nearly identical concepts, but a different presentation)

- Course Notes (by Prof Alan Mycroft and his predecessors). Notes do not reflect changes to lectures. Examinable concepts are those presented in lecture.

Main textbook(s)

- Compiler Design in Java/C/ML (3 books). Andrew W. Appel. (1996)

Other books of interest

- Compilers --- Principles, Techniques, and Tools. Aho, Sethi, and Ullman (1986)
- Compiler Design. Wilhelm, Maurer (1995)
- A Retargetable C Compiler: Design and Implementation. Frazer, Hanson (1995)
- Compiler Construction. Waite, Goos (1984)
- High-level Languages and Their Compilers. Watson (1989)

LECTURE 2
Slang.1 front-end

- Simple lexical analysis
- The problem of ambiguity
- A hand-written “lexer”
- Context free grammars, parse trees
- The problem of ambiguity
- Rewriting a CFG to avoid ambiguity (when lucky)
- Recursive descent parsing
- Rewriting a CFG to allow recursive descent parsing (eliminating left-recursion)
- Simple type checking

You don’t have to learn LEX and YACC to write a front –end !!!

Slang.1 is verbose syntax for L1 (SPL)

```
datatype type_expr =
  Teint
| Teunit
| TEbool

type loc = string

datatype oper = Plus | Mult | Subt | GTEQ

datatype unary_oper = Neg | Not

datatype expr =
  Skip
| Integer of int
| Boolean of bool
| UnaryOp of unary_oper * expr
| Op of expr * oper * expr
| Assign of loc * (type_expr option) * expr
| Deref of loc
| Seq of expr * expr
| If of expr * expr * expr
| While of expr * expr
| Print of (type_expr option) * expr
```

```
% print the first ten squares
begin
 set n := 10;
 set x := 1;
 while n >= x do
 begin
  print (x * x);
  set x := x + 1
 end
end
```

examples/squares.slang

Parse

An expression of type expr (AST is pretty printed!)

```
n := 10;
x := 1;
while (!n >= !x) do
  (print(!x * !x);
   x := !x + 1)
```

This is the AST of L1 (SPL) with minor modifications noted in red.

Concrete syntax of Slang.1 is designed to make recursive descent parsing easy …

L-values vs. R-values

(in C)

```
x = x + 3;
```

An L-value represents a memory location.

An R-value represents the value stored at the memory location associated with x

The concrete syntax of Slang.1 uses this C-like notation, while the AST (in L1) produced by the front end uses !x to represent the R-value associated with L-value x.

In C and Slang.3 L-values may be determined at run-time:

```
A[j*2] = j + 3;
```

(C example)

Slang.1 lexical matters (informal)

- **Keywords:** begin end if then else set while do skip print true false
- **Identifiers:** starting with A-Z or a-z, followed by zero or more characters in A-Z, a-z, or 0-9
- **Integer constants:** starting with 0-9 followed by zero or more characters in 0-9
- **Special symbols:** + * - ~ ; := >= ( )
- **Whitespace:** tabs, space, newline, comments start anywhere with a "%" and consume the remainder of the line

Ambiguity must be resolved

- **Priority:** the character sequence "then" could be either an identifier or a keyword. We declare that keywords win.
- **Longest Match:** example: "xy" is a single identifier, not two identifiers "x" and "y".

From Character Streams to Token Streams

```
datatype token =
     Teof                (* end-of-file *)
   | Tint of int         (* integer     *)
   | Tident of string    (* identifier  *)
   | Ttrue               (* true        *)
   | Tfalse              (* false       *)
   | Tright_paren        (* )           *)
   | Tleft_paren         (* (           *)
   | Tsemi               (* ;           *)
   | Tplus               (* +           *)
   | Tstar               (* *           *)
   | Tminus              (* -           *)
   | Tnot                (* ~           *)
   | Tgets               (* :=          *)
   | Tgteq               (* >=          *)
   | Tset                (* set         *)
   | Tskip               (* skip        *)
   | Tbegin              (* begin       *)
   | Tend                (* end         *)
   | Tif                 (* if          *)
   | Tthen               (* then        *)
   | Telse               (* else        *)
   | Twhile              (* while       *)
   | Tdo                 (* do          *)
   | Tprint              (* print       *)
```

```
% print the first ten squares
begin
 set n := 10;
 set x := 1;
 while n >= x do
 begin
  print (x * x);
  set x := x + 1
 end
end
```

examples/squares.slang

LEX

```
Tbegin, Tset, Tident "n", Tgets, Tint 10,
Tsemi, Tset, Tident "x", Tgets, Tint 1,
Tsemi, Twhile, Tident "n", Tgteq, Tident
"x", Tdo, Tbegin, Tprint,Tleft_paren,
Tident "x", Tstar, Tident "x",
Tright_paren, Tsemi, Tset, Tident "x",
Tgets, Tident "x", Tplus, Tint 1, Tend,
Tend, Teof
```

Note that white-space has vanished. Don't try that with Python or with http://compsoc.dur.ac.uk/whitespace/

A peek into CompilerConstruction2013/slang/slang.1/Lexer.sml

```
exception  LexerError of string;

datatype token =
     Teof                  (* end-of-file *)
   | Tint of int           (* integer     *)
   | Tident of string      (* identifier  *)

   …
   … see previous slide …
   …

type lex_buffer

val init_lex_buffer    : string -> lex_buffer (* string is filename *)
val peek_next_token    : lex_buffer -> token
val consume_next_token : lex_buffer -> (lex_buffer * token)
```

The lexer interface as seen by the parser.

A few implementation details

```
datatype lex_buffer = LexBuffer of {
  lexBuffer : string, (* the entire input file! *)
  lexPosition : int,
  lexSize : int
}
fun consume_next_token lex_buf =
    let val lex_buf1 = ignore_whitespace lex_buf
    in
        if at_eof lex_buf1
       then (lex_buf1, Teof)
       else get_longest_match lex_buf1
    end

fun peek_next_token lex_buf =
    let val lex_buf1 = ignore_whitespace lex_buf
    in
        if at_eof lex_buf1
       then Teof
       else let val (_, tok) = get_longest_match lex_buf1 in tok end
    end
```

A few implementation details

```
fun ignore_comment lex_buf =
   if at_eof lex_buf
   then lex_buf
   else case current_char lex_buf of
          #"\n" => ignore_whitespace (advance_pos 1 lex_buf)
        | _     => ignore_comment (advance_pos 1 lex_buf)

and ignore_whitespace lex_buf =
   if at_eof lex_buf
   then lex_buf
   else case current_char lex_buf of
          #" "  => ignore_whitespace (advance_pos 1 lex_buf)
        | #"\n" => ignore_whitespace (advance_pos 1 lex_buf)
        | #"\t" => ignore_whitespace (advance_pos 1 lex_buf)
        | #"%"  => ignore_comment    (advance_pos 1 lex_buf)
        | _     => lex_buf
```

Later in the term we will see how to generate code for lexical analysis from a specification based on Regular Expressions (how LEX works)

On to Context Free Grammars

E ::= ID

E ::= NUM

E ::= E * E

E ::= E / E

E ::= E + E

E ::= E – E

E ::= (E)

E is a *non-terminal symbol*

ID and NUM are *lexical classes*

*, (,), +, and – are *terminal symbols*.

E ::= E + E is called a *production rule*.

Usually will write this way

E ::= ID | NUM | E * E | E / E | E + E | E – E | (E)

Grammar Derivations

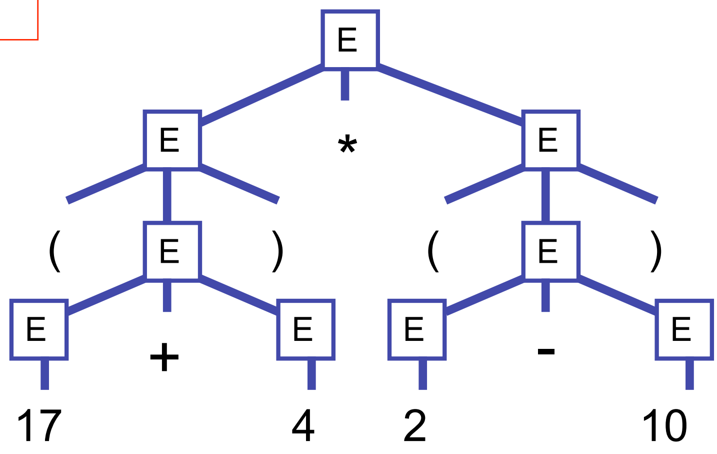
(G1) E ::= ID | NUM | ID | E * E | E / E | E + E | E – E | (E)

Rightmost derivation

E → E * E
→ E * (E)
→ E * (E – E)
→ E * (E – 10)
→ E * (2 – 10)
→ (E) * (2 – 10)
→ (E + E) * (2 – 10)
→ (E + 4) * (2 – E)
→ (17 + 4) * (2 – 10)

Leftmost derivation

E → E * E
→ (**E**) * E
→ (**E** + E) * E
→ (17 + E) * E
→ (17 + 4) * E
→ (17 + 4) * (E)
→ (17 + 4) * (**E** – E)
→ (17 + 4) * (2 – **E**)
→ (17 + 4) * (2 – 10)



The Derivation Tree for
(17 + 4) * (2 – 10)

More formally, …

- **A Context Free Grammar is a quadruple G = (N, T, R, S) where**
  - **N is the set of *non-terminal symbols***
  - **T is the set of *terminal symbols* (N and T disjoint)**
  - **S $\in$N is the *start symbol***
  - **R $\subseteq$ N$\times$(N$\cup$T)* is a set of rules**
- **Example: The grammar of nested parentheses G = (N, T, R, S) where**
  - **N = {S}**
  - **T ={ (,) }**
  - **R ={ (S, (S)) , (S, SS), (S,) }**

We will normally write R as S ::= (S) | SS |

Derivations, more formally…

- **Start from start symbol (*S*)**
- **Productions are used to derive a sequence of tokens from the start symbol**
- **For arbitrary strings α, β and γ comprised of both terminal and non-terminal symbols,**
  and a production A $\rightarrow \beta$,
  a single step of derivation is
  $\alpha A \gamma \Rightarrow \alpha\beta\gamma$
  – ***i.e.*, substitute β for an occurrence of A**
- **$\alpha \Rightarrow^* \beta$ means that b can be derived from a in 0 or more single steps**
- **$\alpha \Rightarrow^+ \beta$ means that b can be derived from a in 1 or more single steps**

L(G) = The Language Generated by Grammar G

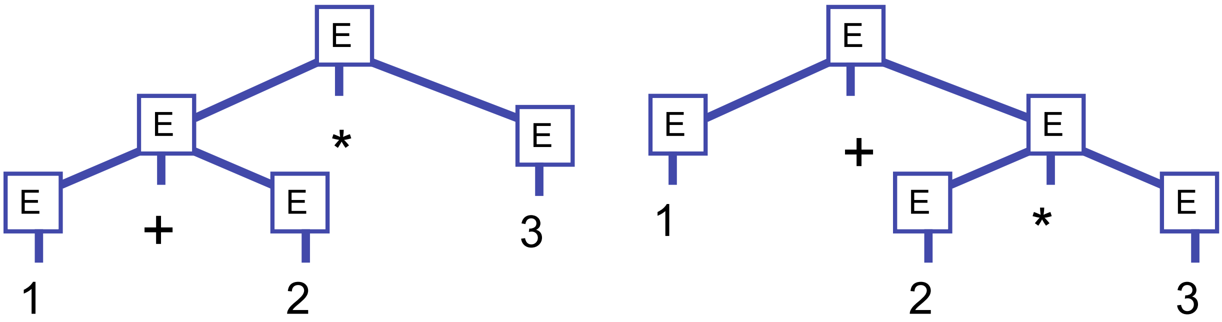
The language generated by G is the set of all terminal strings derivable from the start symbol S:

$$L(G) = \{w \in T^* \mid S \Rightarrow +w\}$$

For any subset W of T*, if there exists a Context Free Grammar G such that L(G) = W, then W is called a Context-Free Language over T.

Ambiguity

(G1) E ::= ID | NUM | ID | E * E | E / E | E + E | E – E | (E)



Both derivation trees correspond to the string

1 + 2 * 3

This type of ambiguity will cause problems when we try to go from strings to derivation trees!

Problem: Generation vs. Parsing

- **Context-Free Grammars (CFGs) describe how to to *generate***
- ***Parsing* is the inverse of generation,**
  - **Given an input string, is it in the language generated by a CFG?**
  - **If so, construct a derivation tree (normally called a *parse tree*).**
  - **Ambiguity is a big problem**

Note : recent work on Parsing Expression Grammars (PEGs) represents an attempt to develop a formalism that describes parsing directly. This is beyond the scope of these lectures …

We can often modify the grammar in order to eliminate ambiguity

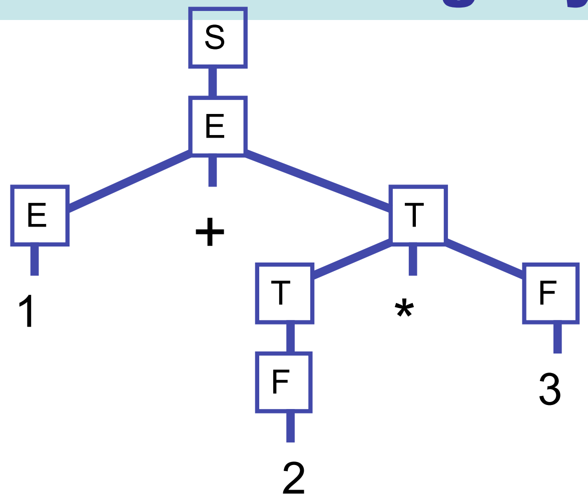
(G2)
S :: = E$ — (start, $ = EOF)

E ::= E + T
| E – T
| T — (expressions)

T ::= T * F
| T / F
| F — (terms)

F ::= NUM
| ID
| (E) — (factors)

Note: L(G1) = L(G2). Can you prove it?



This is the *unique* derivation tree for the string

1 + 2 * 3$

Famously Ambiguous

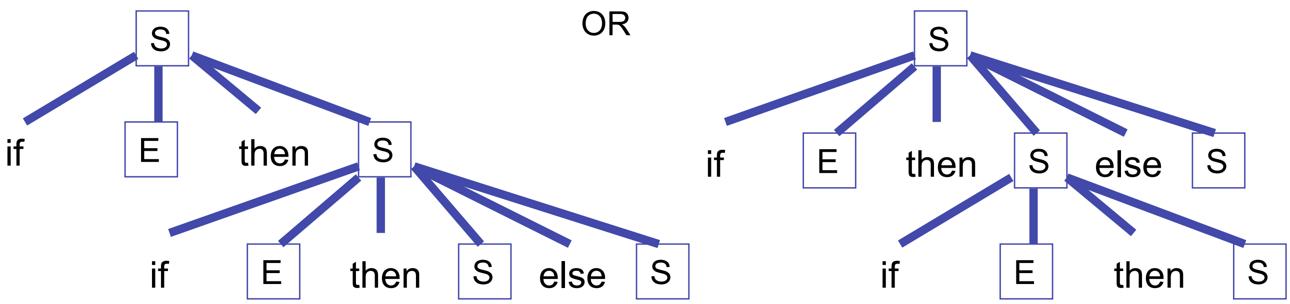
(G3) S ::= if E then S else S | if E then S | blah-blah

What does

if e1 then if e2 then s1 else s3

mean?

OR



Rewrite?

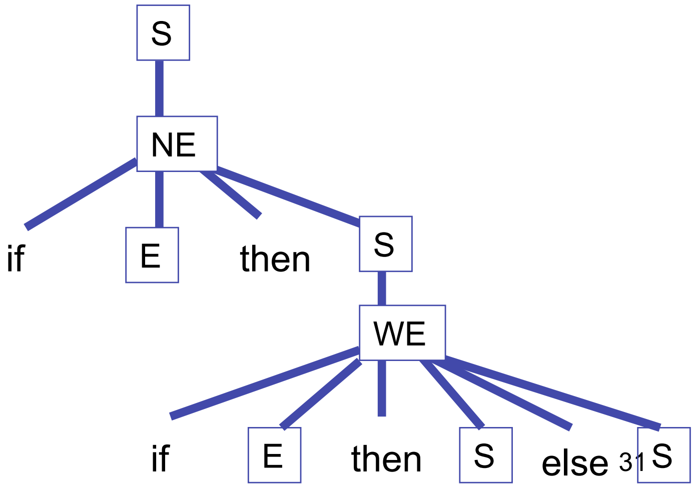
(G4)
S ::= WE | NE
WE ::= **if** E **then** WE **else** WE | **blah-blah**
NE ::= **if** E **then** S
| **if** E **then** WE **else** NE

Now,

if e1 then if e2 then s1 else s3

has a unique derivation.

Note: L(G3) = L(G4).
Can you prove it?



Fun Fun Facts

See Hopcroft and Ullman, "Introduction to Automata Theory, Languages, and Computation"

(1) Some context free languages are *inherently ambiguous* --- every context-free grammar will be ambiguous. For example:

$$L = \{a^n b^n c^m d^m \mid m \geq 1, n \geq 1\} \cup \{a^n b^m c^m d^n \mid m \geq 1, n \geq 1\}$$

(2) Checking for ambiguity in an arbitrary context-free grammar is not decidable! Ouch!

(3) Given two grammars G1 and G2, checking L(G1) = L(G2) is not decidable! Ouch!

Recursive Descent Parsing

(G5)

S :: = if E then S else S
| begin S L
| print E

E ::= NUM = NUM

L ::= end
| ; S L

From Andrew Appel, "Modern Compiler Implementation in Java" page 46

```
int tok = getToken();

void advance() {tok = getToken();}
void eat (int t) {if (tok == t) advance(); else error();}

void S() {switch(tok) {
      case IF:    eat(IF); E(); eat(THEN);
                  S(); eat(ELSE); S(); break;
      case BEGIN: eat(BEGIN); S(); L(); break;
      case PRINT: eat(PRINT); E(); break;
      default: error();
     }}

void L() {switch(tok) {
      case END:  eat(END); break;
      case SEMI: eat(SEMI); S(); L(); break;
      default: error();
     }}

void E() {eat(NUM) ; eat(EQ); eat(NUM); }
```

Parse corresponds to a left-most derivation constructed in a "top-down" manner

PROBLEM : "left recursive grammars" such as G2 (E ::= E + T | E – T | T) will cause code based on this method to go into an infinite loop!

Rewrite grammar to eliminate left recursion

(G2)
S :: = E$

E ::= E + T
| E – T
| T

T ::= T * F
| T / F
| F

F ::= NUM
| ID
| (E)

Eliminate left recursion →

(G6)
S :: = E$

E ::= T E'

E' ::= + T E'
| – T E'
|

T ::= F T'

T' ::= * F T'
| / F T'
|

F ::= NUM
| ID
| (E)

Note: L(G2) = L(G6). Can you prove it?

Finally, our Slang.1 grammar

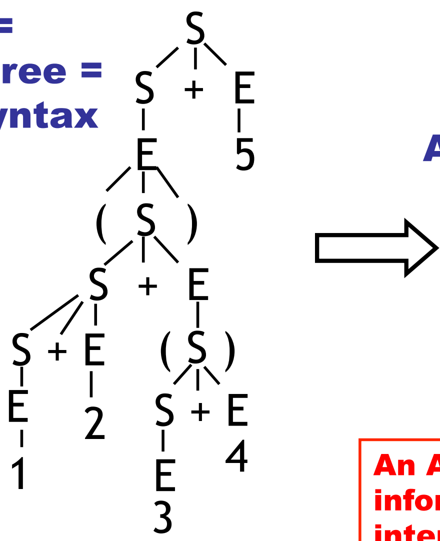
```
program := expr EOF

expr := simple
| set identifier := expr
| while expr do expr
| if expr then expr else expr
| begin expr expr_list

expr_list := ; expr expr_list
           | end

simple ::= term srest

term ::= factor trest
```

```
srest ::=  + term srest
        |  – term srest
        |  >= term srest
        |

trest ::=  ∗  factor trest
           |

factor := identifier
| integer
| – expr
| true
| false
| skip
| ( expr )
| print expr
```

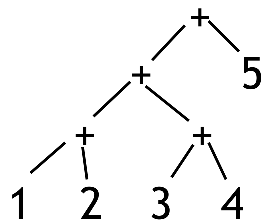
The grammar has been designed to avoid ambiguity and to make recursive descent parsing very very easy

Concrete vs. Abstract Syntax Trees

parse tree = derivation tree = concrete syntax tree



Abstract Syntax Tree (AST)



An AST contains only the information needed to generate an intermediate representation

Normally a compiler constructs the concrete syntax tree only implicitly (in the parsing process) and explicitly constructs an AST.

A peek at slang.1/parser.sml

```
expr := simple
| set identifier := expr
| while expr do expr
| if expr then expr else expr
| begin expr expr_list
```

```
fun parse_expr lex_buf =
    let val (lex_buf1, next_token) = consume_next_token lex_buf
    in case next_token of
           Tset    => let val (lex_buf2, id) = parse_id lex_buf1
                          val lex_buf3 = parse_gets lex_buf2
                          val (lex_buf4, e) = parse_expr lex_buf3
                      in (lex_buf4, Assign(id, NONE, e)) end
         | Twhile => let val (lex_buf2, e1) = parse_expr lex_buf1
                          val lex_buf3 = parse_do lex_buf2
                          val (lex_buf4, e2) = parse_expr lex_buf3
                      in (lex_buf4, While(e1, e2)) end
         | Tif    => let val (lex_buf2, e1) = parse_expr lex_buf1
                          val lex_buf3 = parse_then lex_buf2
                          val (lex_buf4, e2) = parse_expr lex_buf3
                          val lex_buf5 = parse_else lex_buf4
                          val (lex_buf6, e3) = parse_expr lex_buf5
                      in (lex_buf6, If(e1, e2, e3)) end
         | Tbegin => let val (lex_buf2, e1) = parse_expr lex_buf1
                          val (lex_buf3, e_opt) = parse_expr_list lex_buf2
                      in case e_opt of
                            SOME e2 => (lex_buf3, Seq(e1, e2))
                          | NONE    => (lex_buf3, e1)
                      end
         | _      => parse_simple lex_buf
    end
```


Types : SPL give us the rules

(int) $\Gamma \vdash n{:}\mathsf{int}$ for $n \in \mathbb{Z}$

(bool) $\Gamma \vdash b{:}\mathsf{bool}$ for $b \in \{\mathsf{true}, \mathsf{false}\}$

(op +) $$\frac{\Gamma \vdash e_1{:}\mathsf{int} \quad \Gamma \vdash e_2{:}\mathsf{int}}{\Gamma \vdash e_1 + e_2{:}\mathsf{int}}$$

(op $\geq$) $$\frac{\Gamma \vdash e_1{:}\mathsf{int} \quad \Gamma \vdash e_2{:}\mathsf{int}}{\Gamma \vdash e_1 \geq e_2{:}\mathsf{bool}}$$

(if) $$\frac{\Gamma \vdash e_1{:}\mathsf{bool} \quad \Gamma \vdash e_2{:}T \quad \Gamma \vdash e_3{:}T}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3{:}T}$$

(assign) $$\frac{\Gamma(\ell) = \mathsf{intref} \quad \Gamma \vdash e{:}\mathsf{int}}{\Gamma \vdash \ell := e{:}\mathsf{unit}}$$

(deref) $$\frac{\Gamma(\ell) = \mathsf{intref}}{\Gamma \vdash !\ell{:}\mathsf{int}}$$

(skip) $\Gamma \vdash \mathsf{skip}{:}\mathsf{unit}$

(seq) $$\frac{\Gamma \vdash e_1{:}\mathsf{unit} \quad \Gamma \vdash e_2{:}T}{\Gamma \vdash e_1; e_2{:}T}$$

(while) $$\frac{\Gamma \vdash e_1{:}\mathsf{bool} \quad \Gamma \vdash e_2{:}\mathsf{unit}}{\Gamma \vdash \mathsf{while}\ e_1\ \mathsf{do}\ e_2{:}\mathsf{unit}}$$

But wait! Where can we find Γ (gamma)? We must construct it from the program text. How?

Note : details of SPL material are of course not examinable in CC questions!

SPL give us an option …

Slide 38

Language design 3. Store initialization

Recall that

(deref) $\langle !\ell, s\rangle \longrightarrow \langle n, s\rangle$ if $\ell \in \mathrm{dom}(s)$ and $s(\ell) = n$

(assign1) $\langle \ell := n, s\rangle \longrightarrow \langle \mathsf{skip}, s + \{\ell \mapsto n\}\rangle$ if $\ell \in \mathrm{dom}(s)$

both require $\ell \in \mathrm{dom}(s)$, otherwise the expressions are stuck.

Instead, could

1. implicitly initialize *all* locations to 0, or
2. allow assignment to an $\ell \notin \mathrm{dom}(s)$ to initialize that ℓ.

We like the first option!

Yes, these are not typing rules but rules of operational semantics

One possible interpretation of option 2: If when evaluating an expression we encounter a "!x", then we must have previously evaluated a "x := v" for some integer v.

Oh bother, that's a dynamic notion where the program to the right is correct …

```
begin
 set n := 10;
 if (n+n) >= (2*n)
 then print n
 else print x
end
```

In later versions of the language these issues are cleanly resolved by well-structured scope and declaration rules …

check static semantics

```
fun check_static_semantics e = let val (_, e') = ccs e in e' end

css : expr -> (type_expr * expr)
```

```
…
css env (If (e1,e2,e3)) =
     let val (t1, e1') = css e1
         val (t2, e2') = css e2
         val (t3, e3') = css e3
     in
        if t1 = TEbool
        then if t2 = t3
             then (t2, If (e1', e2', e3'))
             else type_error … …
        else type_error  … …
     end
…
```

$$(\text{if})\ \frac{\Gamma \vdash e_1{:}\mathsf{bool} \quad \Gamma \vdash e_2{:}T \quad \Gamma \vdash e_3{:}T}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3{:}T}$$

Theorem: if

(t, e') = css e

Then

|- e : t

and erase(e') = e, where erase removes all type annotations.

Prove by induction on the structure of e.

Not so interesting in Slang.1, but later ….

Front-end example: squares.slang

```
begin
 set n := 10;
 set x := 1;
 while n >= x do
 begin
  print (x * x);
  set x := x + 1
 end
end
```

lex and parse →

```
n := 10;
x := 1;
while (!n >= !x) do
  (
   print(!x * !x);
   x := !x + 1
  )
```

check static semantics →

```
n : int := 10;
x : int := 1;
while (!n >= !x) do
  (
   print(int,!x * !x);
   x :int := !x + 1
  )
```

Next two lectures : translating output of front-end into bytecodes for two virtual machines

LECTURES 3 & 4
Targeting Virtual Machines

- **Register-oriented vs Stack-oriented virtual machines**
- **Computation in registers requires arguments to have a location**
- **Computation at the "top of the stack" allows arguments to be implicit**
- **Otherwise, compilation of control constructs (if-then-else, while, sequence) looks very similar**
- **For Slang.1 the L1 semantics keeps us more-or-less honest**
- **Simple "peep-hole" optimization**

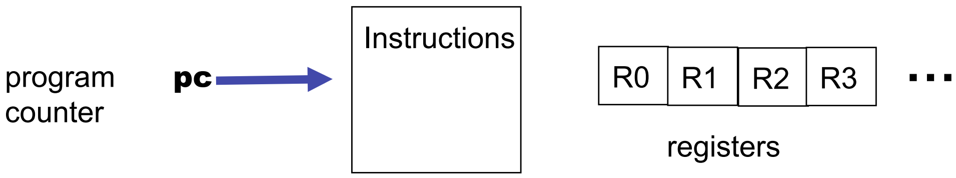
By the end of lecture 4 you will understand a complete compiler for Slang.1 targeting two virtual machines.
Yes, the language is very simple at this point …

A word about Virtual Machines

- Martin Richards (Cambridge) define a virtual machine for BCPL in the late 1960s.
- Virtual machines allow greater portability
- Virtual machines enable “sand boxing” --- isolating the host system from potentially malicious code
- JVM originally designed for set-top boxes
- JVM is stack-oriented
- Dalvik is the VM of Android
- Dalvik is register-oriented

- Of course there is a performance cost in using a VM compared to a ISA/OS

Virtual Register Machine (VRM.0)




```
nop            : pc <- !pc +1
set r c        : r <- c           ; pc <- !pc +1
mov r1 r2      : r1 <- !r2        ; pc <- !pc +1
add r1 r2 r3 : r1 <- !r1 + !r2 ; pc <- !pc +1
sub r1 r2 r3 : r1 <- !r1 - !r2 ; pc <- !pc +1
mul r1 r2 r3 : r1 <- !r1 * !r2 ; pc <- !pc +1
hlt            : halt the machine
jmp l          : pc <- l
ifz r l        : if !r == 0 then pc <- l else pc <- !pc+1
ifp r l        : if !r >= 0 then pc <- l else pc <- !pc+1
ifn r l        : if !r <  0 then pc <- l else pc <- !pc+1
pri r          : prints out !r as an integer; pc <- !pc+1
```

Instruction set. The notation “!r” means the contents of register r and “<-” is assignment.

Byte code instructions as stored in object files

opcode	arg1	arg2	arg2	
1 byte				`hlt, nop`
1 byte	1 byte			`jmp, pri`
1 byte	1 byte	1 byte		`if_, set, mov`
1 byte	1 byte	1 byte	1 byte	`add, mul, sub`

Object file = 1 byte version (0) + 1 byte instruction count + sequence of bytecode instructions

A tiny machine! At most 256 instructions per program and no more than 256 registers….

About VRM.0 implementation

```
void vrm_execute_instruction(vrm_state *state, bytecode instruction)
{
  opcode code   = instruction.code;
  argument arg1 = instruction.arg1;
  argument arg2 = instruction.arg2;
  argument arg3 = instruction.arg3;

  switch (code) {
       case OP_NOP:
        {
            state->pc++;
            break;
        }
        case OP_SET:
        {
            state->registers[arg1] = arg2;
            state->touched[arg1] = 1; /* used in verbose mode */
            state->pc++;
            break;
        }
        case OP_MOV:
        {
            state->registers[arg1] = state->registers[arg2];
            state->touched[arg1] = 1;
            state->touched[arg2] = 1;
            state->pc++;
            break;
       …
       …
```

Very simple:

about 400 lines of C

Very tiny:

No more than 256 instructions per program

“Only” 256 registers

Only 13 basic Instructions

Efficiency of C code could be improved dramatically

From slang/slang.1/AST_vrm_assembler.sml

```
type vrm_data_loc = string  (* symbolic, not numeric! *)
type vrm_code_loc = string  (* symbolic, not numeric! *)
type vrm_constant = int
type vrm_comment = string (* for instructional purposes! *)

datatype vrm_operation =
       (* data operations *)
         VRM_Nop of vrm_comment
       | VRM_Set of vrm_data_loc * vrm_constant * vrm_comment
       | VRM_Mov of vrm_data_loc * vrm_data_loc * vrm_comment
       | VRM_Add of vrm_data_loc * vrm_data_loc * vrm_data_loc * vrm_comment
       | VRM_Sub of vrm_data_loc * vrm_data_loc * vrm_data_loc * vrm_comment
       | VRM_Mul of vrm_data_loc * vrm_data_loc * vrm_data_loc * vrm_comment
       (* control flow operations *)
       | VRM_Hlt of vrm_comment
       | VRM_Jmp of vrm_code_loc * vrm_comment
       | VRM_Ifz of vrm_data_loc * vrm_code_loc * vrm_comment
       | VRM_Ifp of vrm_data_loc * vrm_code_loc * vrm_comment
       | VRM_Ifn of vrm_data_loc * vrm_code_loc * vrm_comment
       (* input/output *)
       | VRM_Pri of vrm_data_loc * vrm_comment

datatype vrm_code =
         VRM_Code of vrm_operation
       | VRM_Labelled of vrm_code_loc * vrm_operation

type vrm_assembler = vrm_code list
```


Mind the Gap
--- two main issues ---

L1 (output of front-end)	VRM.0 programs
One of FORTRAN's major innovations --- "unnamed" sub-expression `3 * ((8 + 17) * (2 - 6))`	set r0 3 set r1 8 set r2 17 add r3 r1 r2 set r4 2 set r5 6 sub r6 r4 _X5 mul r7 r3 _X6 mul r8 r0 r7 Operations only on "named" registers (Not Optimal!)
Structured control operations, If-then-else, while-do	Unstructured control operations, jmp, if_

Bridging the Gap

Our Slang.1 compiler bridges the gap by first "naming" every sub-expression and then eliminating structured control.

One of the "Slang.1 Programming Exercises" leads you to question the wisdom of this choice.

Think about eliminating structured control first … Try to implement this. Best solution will win a Kit-Kat bar!

normalise

AST_expr.sml

```
datatype expr =
  Skip
| Integer of int
| Boolean of bool
| UnaryOp of unary_oper * expr
| Op of expr * oper * expr
| Assign of loc * (type_expr option) * expr
| Deref of loc
| Seq of expr * expr
| If of expr * expr * expr
| While of expr * expr
| Print of (type_expr option) * expr
```

AST_normal_expr.sml

```
datatype normal_expr =

  Normal_SetInteger of loc * int
| Normal_SetBoolean of loc * bool
| Normal_UnaryOp of unary_oper * loc * loc
| Normal_Op of oper * loc * loc * loc
| Normal_Assign of loc * loc

| Normal_Seq of normal_expr list
| Normal_If of into_expr * normal_expr * normal_expr
| Normal_While of into_expr * normal_expr
| Normal_Print of (type_expr option) * loc

and into_expr = Into of normal_expr * loc
```

`normalise : expr -> normal_expr`

The datatype `normal_expr` forces every intermediate value to be stored in a named location. Conditionals and loops must know where to their find test value, thus `into_expr`.

normalise

```
fun normalise e =
    let val (el, _) = normalise_expr e
        and init_code = locs_to_init_code ("_Unit" :: (all_locs [] e))
    in
      Normal_Seq (init_code @ el)
    end
```

Code to initialize all used locations to 0

```
normalise_expr :expr -> ((normal_expr list )* loc)
```

The idea: if

(el, l) = normalise_expr e

then evaluating the sequence el will leave a value in location l. This is the same value obtained by evaluating e.

To formalize this we would have to give a semantics to normal_expr expressions.

I hope we can leave it informal …

HA! Another slide with "Mid-Atlantic " spelling!

normalise_expr --- easy bits

```
fun normalise_expr Skip = ([], "_Unit")
  | normalise_expr (Integer n) =
    let val l = new_location()
    in
       ([Normal_SetInteger(l, n)], l)
    end
  | normalise_expr (Boolean b) =
    let val l = new_location()
    in
       ([Normal_SetBoolean(l, b)], l)
    end
  | normalise_expr (UnaryOp (uop, e)) =
    let val (el, l) = normalise_expr e
        and l' = new_location()
    in
        (el @ [Normal_UnaryOp(uop, l', l)], l')
    end
 | normalise_expr (Assign (l, _, e)) =
    let val (el, l') = normalise_expr e
    in
       (el @ [Normal_Assign(l, l')], "_Unit")
    end
  | normalise_expr (Deref l) = ([], l)

… …
```

normalise_expr --- tricky bit

```
… …
normalise_expr (Op (bop, e1, e2)) =
    let val (el1, l1) = normalise_expr e1
        and (el2, l2) = normalise_expr e2
        and l3 = new_location()
    in
       (el1 @ el2 @ [Normal_Op(bop, l3, l1, l2)], l3)
    end
… …
```

Is this Correct?

No!

Counter example:

```
% should print "-40"
  begin
       set x := 10 ;
       set x := (begin set x := 4 * x ; x end)
                  - (begin set x := 2 * x ; x end);
       print x
  end
```

Problem : running el2 could change the value "saved" in l1

normalise_expr --- tricky bit, solved

```
… …
normalise_expr (Op (bop, e1, e2)) =
    let val (el1, l1) = normalise_expr e1
        and (el2, l2) = normalise_expr e2
        and l3 = new_location()
    in
       if can_update(l1, e2)
       then let val l4 = new_location()
            in
               (el1 @ [Normal_Assign(l4, l1)] @ el2 @ [Normal_Op(bop, l3, l4, l2)], l3)
            end
       else (el1 @ el2 @ [Normal_Op(bop, l3, l1, l2)], l3)
    end
… …
```

can_update(l, e) is true when evaluating e could change the value stored at l

```
fun can_update (l, UnaryOp (_, e))   = can_update(l, e)
  | can_update (l, Op (_,e1,e2))     = (can_update(l, e1)) orelse (can_update(l, e2))
  | can_update (l, If (e1, e2, e3))  = (can_update(l, e1)) orelse (can_update(l, e2))
                                                            orelse (can_update(l, e3))
  | can_update (l, Assign (l',_, e)) = (l = l') orelse (can_update(l, e))
  | can_update (l, Seq (e1,e2))      = (can_update(l, e1)) orelse (can_update(l, e2))
  | can_update (l, While (e1,e2))    = (can_update(l, e1)) orelse (can_update(l, e2))
  | can_update (l, Print (_, e))     = can_update(l, e)
  | can_update _                     = false
```

vrm_code_gen

```
datatype normal_expr =

  Normal_SetInteger of loc ∗ int
| Normal_SetBoolean of loc ∗ bool
| Normal_UnaryOp of unary_oper ∗ loc ∗ loc
| Normal_Op of oper ∗ loc ∗ loc ∗ loc
| Normal_Assign of loc ∗ loc

| Normal_Seq of normal_expr list
| Normal_If of into_expr ∗ normal_expr ∗ normal_expr
| Normal_While of into_expr ∗ normal_expr
| Normal_Print of (type_expr option) ∗ loc

and into_expr = Into of normal_expr ∗ loc
```

```
datatype vrm_operation =
  VRM_Nop of vrm_comment
| VRM_Set of vrm_data_loc ∗ vrm_constant ∗ v
| VRM_Mov of vrm_data_loc ∗ vrm_data_loc ∗ v
| VRM_Add of vrm_data_loc ∗ vrm_data_loc ∗ v
| VRM_Sub of vrm_data_loc ∗ vrm_data_loc ∗ v
| VRM_Mul of vrm_data_loc ∗ vrm_data_loc ∗ v
| VRM_Hlt of vrm_comment
| VRM_Jmp of vrm_code_loc ∗ vrm_comment
| VRM_Ifz of vrm_data_loc ∗ vrm_code_loc ∗ v
| VRM_Ifp of vrm_data_loc ∗ vrm_code_loc ∗ v
| VRM_Ifn of vrm_data_loc ∗ vrm_code_loc ∗ v
| VRM_Pri of vrm_data_loc ∗ vrm_comment
| VRM_Prb of vrm_data_loc ∗ vrm_comment

datatype vrm_code =
         VRM_Code of vrm_operation
       | VRM_Labelled of vrm_code_loc ∗ vrm_

type vrm_assembler = vrm_code list
```

```
vrm_code_gen : normal_expr -> vrm_assembler
```

We need only eliminate structured control and implement the binary/unary operations not directly provided by VRM.0

Code generation

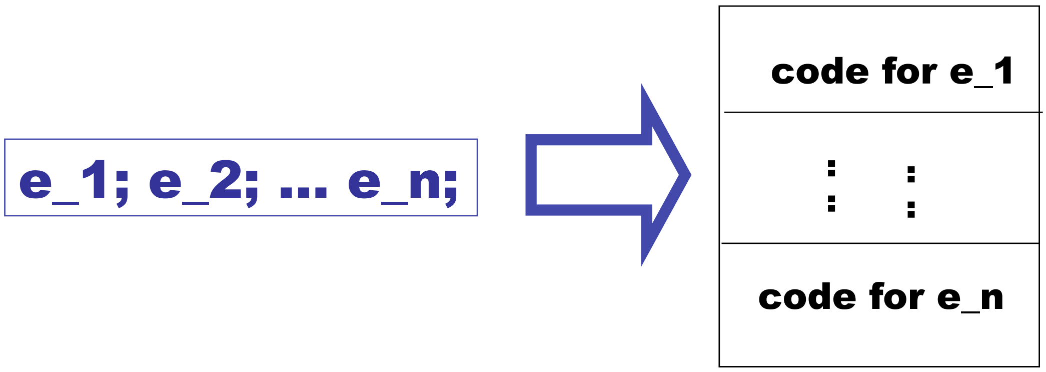
vrm_code_gen : normal_expr -> vrm_assembler

```
val zero_loc   = "_Zero"
val true_loc   = "_TRUE"
val false_loc  = "_FALSE”

val init_code =
   [VRM_Code(VRM_Set(zero_loc, 0, " zero")),
    VRM_Code(VRM_Set(true_loc, 1, " true value")),
    VRM_Code(VRM_Set(false_loc, 0, " false value"))]

fun normal_expr_to_vrm_code_list (Normal_SetInteger (l, n)) =
       [VRM_Code(VRM_Set(l, n, ""))]
       …..
       …..
```

```
fun vrm_code_gen e =
    init_code
    @ (normal_expr_to_vrm_code_list e)
    @ [VRM_Code (VRM_Hlt " that's all folks!")]
```

List.@ : 'a list * 'a list -> 'a list

Sequence is easy!



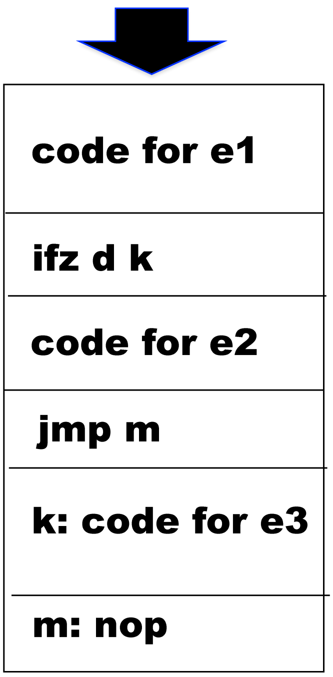

```
| normal_expr_to_vrm_code_list (Normal_Seq el)  =
    List.concat (List.map normal_expr_to_vrm_code_list el)
```

Remember these?

```
List.concat :  'a list list -> 'a list

List.map :  : ('a -> 'b) -> 'a list -> 'b list
```

Conditionals

if e1 into d then e2 else e3




```
| normal_expr_to_vrm_code_list (Normal_If(Into(e1, t), e2, e3))  =
   let val cl_cond = normal_expr_to_vrm_code_list e1
       and cl_then = normal_expr_to_vrm_code_list e2
       and cl_else = normal_expr_to_vrm_code_list e3
   in
       let val (l_else, cl_else_new) = vrm_label_sequence cl_else
           and l_end = Library.new_label()
       in
          (vrm_insert_remark "start if (condition)  ... " cl_cond)
          @ [VRM_Code(VRM_Ifz(t, l_else, "test of if ..."))]
          @ (vrm_insert_remark "start then ... " cl_then)
          @ [VRM_Code(VRM_Jmp (l_end, "... end then ..."))]
          @ (vrm_insert_remark "start else ... " cl_else_new)
          @ [VRM_Labelled(l_end, VRM_Nop "... end if")]
       end
   end
```

Loops

while e1 into d do e2

k: code for e1
ifz d m
code for e2
jmp k
m: nop

```
| normal_expr_to_vrm_code_list (Normal_While(Into(e1, d), e2))  =
   let val cl_cond = normal_expr_to_vrm_code_list e1
       and cl_body = normal_expr_to_vrm_code_list e2
       and l_end   = Library.new_label ()
   in
       let val (l_cond, cl_cond_new) = vrm_label_sequence cl_cond
       in
          (vrm_insert_remark "start while ... " cl_cond_new)
          @ [VRM_Code(VRM_Ifz(d, l_end, "test of while ..."))]
          @ cl_body
          @ [VRM_Code(VRM_Jmp (l_cond,
                                          "... go back to while condition ")),
             VRM_Labelled(l_end, VRM_Nop "... end while")]
       end
   end
```


The Slang.1/VRM compiler!

From slang1/slang_compile.sml

```
fun vrm_compile fin fout =
    emit_vrm_bytecode fout
       (vrm_assemble
          (vrm_code_gen
             (normalise
                (check_static_semantics
                   (parse (init_lex_buffer fin))))))
```

```
            normalize       vrm_code_gen        vrm_assemble
expr  ->  expr  ->  normal_expr  ->  vrm_assembler  ->  vrm_bytecode
```

Annotate with types	Give every sub-expression a location	Eliminate structured control and some operations (>=, -, ~)	Replace symbolic locations and code points with numeric machine registers and addresses

Our view of the middle- and back-ends :
a sequence of small transformations

command-line examples

- **slang1 -vrm examples/squares.slang**
  compile squares.slang to VRM.0 to binary object file examples/squares.vrmo
- **slang1 examples/squares.slang**
  same as above (VRM.0 is the default)
- **slang1 -v examples/squares.slang**
  same as above, but with verbose output at each stage of compilation

- **slang1 -vsm examples/squares.slang**
  compile squares.slang to VSM.0 to binary object file examples/squares.vsmo
- **slang1 -v -vsm examples/squares.slang**
  same as above, but with verbose output at each stage of compilation

- **vrm0 examples/squares.vrmo**
  run VRM.0 on bytecode file
- **vrm0 -v examples/squares.vrmo**
  same as above, but with verbose output
- **vrm0 -s examples/squares.vrmo**
  just print the bytecode

- **vsm0 examples/squares.vsmo**
  run VSM.0 on bytecode file
- **vsm0 -v examples/squares.vsmo**
  same as above, but with verbose output
- **vsm0 -s examples/squares.vsmo**
  just print the bytecode

Slang.1 to VRM example (squares.slang)

```
begin
  set n := 10;
  set x := 1;
  while n >= x do
  begin
    print (x * x);
    set x := x + 1
  end
end
```

parse

```
n := 10;
x := 1;
while (!n >= !x) do
  (
    print(!x * !x);
    x := !x + 1
  )
```

Type check

```
n : int := 10;
x : int := 1;
while (!n >= !x) do
  (
    print(int,!x * !x);
    x :int := !x + 1
  )
```

normalise

```
_Unit := 0;
n := 10;
x := 1;
while (_X0 : = !n >= !x)
into _X0 do
(
   _X1 : = !x * !x;
   print(int,  !_X1);
   _X2 := 1;
   _X3 : = !x + !_X2;
   x    := !_X3
)
```

Yes, this code could be improved! See Programming Exercises and Lecture 9 …

Slang.1 to VRM example (squares.slang)

```
_Unit := 0;
n := 10;
x := 1;
while (_X0 : = !n >= !x)
into _X0 do
(
  _X1 : = !x * !x;
  print(int, !_X1);
  _X2 := 1;
  _X3 : = !x + !_X2;
  x   := !_X3
)
```

code gen

```
        set _Zero 0     % zero
        set _TRUE 1     % true value
        set _FALSE 0    % false value
        set _Unit 0     %
        set n 0         %
        set x 0         %
        set n 10        %
        set x 1         %
_l3 :   sub _X0 n x     %start while ...  start >= ...
        ifn _X0 _l0     %
        mov _X0 _TRUE   %get true
        jmp _l1         %
_l0 :   mov _X0 _FALSE  %get false
_l1 :   nop             %... end >=
        ifz _X0 _l2     %test of while ...
        mul _X1 x x     %
        pri _X1         %
        set _X2 1       %
        add _X3 x _X2   %
        mov x _X3       %
        jmp _l3         %... go back to while condition
_l2 :   nop             %... end while
        hlt             % that's all folks!
```

Happy with this?. No? Either complicate code_gen or improve with another pass. Tradeoffs?

Note the implementation of >=. Perhaps we should add more operations to the VM!

Slang.1 to VRM example (squares.slang)

```
        set _Zero 0
        set _TRUE 1
        set _FALSE 0
        set _Unit 0
        set n 0
        set x 0
        set n 10
        set x 1
_l3 :   sub _X0 n x
        ifn _X0 _l0
        mov _X0 _TRUE
        jmp _l1
_l0 :   mov _X0 _FALSE
_l1 :   nop
        ifz _X0 _l2
        mul _X1 x x
        pri _X1
        set _X2 1
        add _X3 x _X2
        mov x _X3
        jmp _l3
_l2 :   nop
        hlt
```

assemble

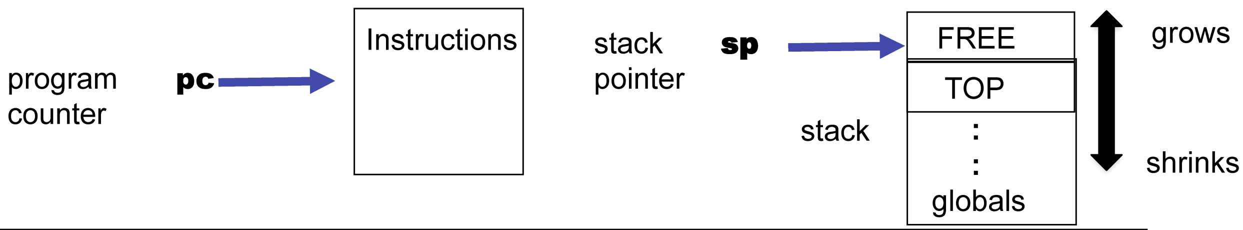
```
l0  : set r0 0
l1  : set r1 1
l2  : set r2 0
l3  : set r3 0
l4  : set r4 0
l5  : set r5 0
l6  : set r4 10
l7  : set r5 1
l8  : sub r6 r4 r5
l9  : ifn r6 l12
l10 : mov r6 r1
l11 : jmp l13
l12 : mov r6 r2
l13 : nop
l14 : ifz r6 l21
l15 : mul r7 r5 r5
l16 : pri r7
l17 : set r8 1
l18 : add r9 r5 r8
l19 : mov r5 r9
l20 : jmp l8
l21 : nop
l22 : hlt
```

Now run it!

```
$ vrm0 examples/squares.vrmo
1
4
9
16
25
36
49
64
81
100
```

```
begin
 set n := 10;
 set x := 1;
 while n >= x do
 begin
  print (x * x);
  set x := x + 1
 end
end
```


Virtual Stack Machine (VSM.0)




```
nop          :                        pc <- !pc +1
push c       : => c                 ; pc <- !pc +1
load m       : => stack[m]          ; pc <- !pc +1
store m      : a => ; stack[m] <- a ; pc <- !pc +1
pop          : a =>                 ; pc <- !pc +1
add          : a, b => a + b        ; pc <- !pc +1
sub          : a, b => b - a        ; pc <- !pc +1
mul          : a, b => a * b        ; pc <- !pc +1
hlt          : HALT the machine
jmp l        : pc <- l
ifz l        : a => ; if a == 0 then pc <- l else pc <- !pc+1
ifp l        : a => ; if a => 0 then pc <- l else pc <- !pc+1
ifn l        : a => ; if a <  0 then pc <- l else pc <- !pc+1
pri          : a => ; print out a as an integer;  pc <- !pc+1
```

Instruction set. The notation “X => Y” means that top of stack is X before operation and Y after.

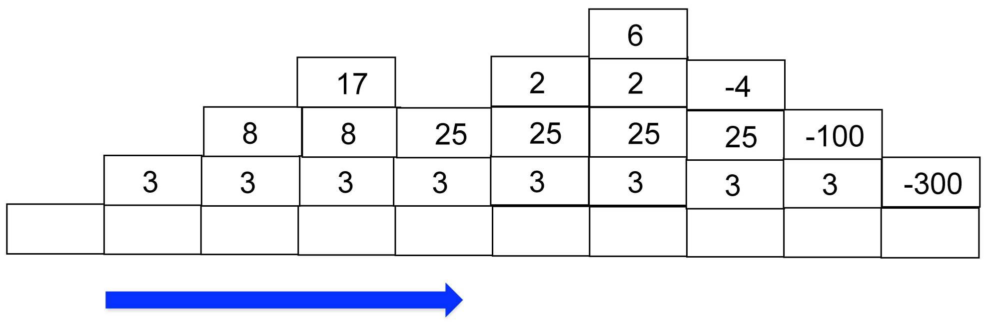
Translation of expressions

e1 op e2

code for e1
code for e2
op

```
3 * ((8 + 17) * (2 - 6))
```

```
push 3
push 8
push 17
add
push 2
push 6
sub
mul
mul
```



Conditional, while loop

if e1 then e2 else e3

code for e1
ifz k
code for e2
jmp m
k: code for e3
m: nop

while e1 do e2

k: code for e1
ifz m
code for e2
jmp k
m: nop

ifz inspects (and consumes) the top-of-stack

Slang.1 to VSM example (back to squares.slang)

Front-end output

```
n : int := 10;
x : int := 1;
while (!n >= !x) do
  (
   print(int,!x * !x);
   x :int := !x + 1
  )
```

code gen →

```
        push 0  %slot for n
        push 0  %slot for x
        push 10 %
        store 0 %store n
        push 0  %push unit value
        pop     %sequence pop
        push 1  %
        store 1 %store x
        push 0  %push unit value
        pop     %sequence pop
_l3 :   load 0  % start while ...  start >= ...  load n
        load 1  %load x
        sub     %
        ifn _l0 %
        push 1  %push true
        jmp _l1 %
_l0 :   push 0  %push false
_l1 :   nop     % ... end <=
        ifz _l2 % test of while ...
        load 1  %load x
        load 1  %load x
        mul     %
        pri     %
        push 0  %push unit value from print
        pop     %sequence pop
        load 1  %load x
        push 1  %
        add     %
        store 1 %store x
        push 0  %push unit value
        pop     %end-of-while-body pop
        jmp _l3 %... jump to while condition
_l2 :   nop     %... end while
        hlt     % that's all folks!
```

"Peep-hole" optimization

Peep hole optimization normally involves sliding a window of some fixed width along a low-level program and replacing various patterns with simpler or more efficient code.

Below is a simple example with window width 2.

```
l: nop
  code    →    l: code
```

```
push c
pop      →
```

```
fun vsm_peep_hole ((VSM_Code(VSM_Push _)) :: ((VSM_Code(VSM_Pop _)) :: rest)) =
       vsm_peep_hole rest
  | vsm_peep_hole ((VSM_Labelled(l, VSM_Nop _)) :: ((VSM_Code(c)) :: rest)) =
       (VSM_Labelled(l, c)) :: (vsm_peep_hole rest)
  | vsm_peep_hole (c :: rest) = c :: (vsm_peep_hole rest)
  | vsm_peep_hole [] = []
```

Sometimes running a peep-hole optimization can create new opportunities for further optimization. Can that happen in our current Slang.1 compiler?

Apply Peep-hole optimization

```
       push 0  %slot for n
       push 0  %slot for x
       push 10 %
       store 0 %store n
       push 0  %push unit value
       pop     %sequence pop
       push 1  %
       store 1 %store x
       push 0  %push unit value
       pop     %sequence pop
_l3 :  load 0  % start while ...   start >= ...   load n
       load 1  %load x
       sub     %
       ifn _l0 %
       push 1  %push true
       jmp _l1 %
_l0 :  push 0  %push false
_l1 :  nop     % ... end <=
       ifz _l2 % test of while ...
       load 1  %load x
       load 1  %load x
       mul     %
       pri     %
       push 0  %push unit value from print
       pop     %sequence pop
       load 1  %load x
       push 1  %
       add     %
       store 1 %store x
       push 0  %push unit value
       pop     %end-of-while-body pop
       jmp _l3 %... jump to while condition
_l2 :  nop     %... end while
       hlt     % that's all folks!
```

```
       push 0  %slot for n
       push 0  %slot for x
       push 10 %
       store 0 %store n


       push 1  %
       store 1 %store x


_l3 :  load 0  % start while ...   start >= ...   load n
       load 1  %load x
       sub     %
       ifn _l0 %
       push 1  %push true
       jmp _l1 %
_l0 :  push 0  %push false

_l1 :  ifz _l2 % test of while ...
       load 1  %load x
       load 1  %load x
       mul     %
       pri     %


       load 1  %load x
       push 1  %
       add     %
       store 1 %store x


       jmp _l3 %... jump to while condition
_l2 :  hlt     % that's all folks!
```

The Slang.1/VSM compiler!

From slang1/slang_compile.sml

```
fun vsm_compile fin fout =
    emit_vsm_bytecode fout
       (vsm_assemble
          (vsm_peep_hole
             (vsm_code_gen
                (check_static_semantics
                   (parse (init_lex_buffer fin))))))
```

```
              vsm_code_gen       vsm_peep_hole        vrm_assemble
expr  ->  expr  ->  vrm_assembler  ->  vsm_assembler  ->  vrm_bytecode
```

Annotate with types

Eliminate structured control and some operations (>=, -, ~)

Apply peep-hole rules

Replace symbolic locations and code points with numeric machine registers and addresses

Our view of the middle- and back-ends : a sequence of small transformations

Now run it!

```
$ vsm0 examples/squares.vsmo
1
4
9
16
25
36
49
64
81
100
```

```
begin
  set n := 10;
  set x := 1;
  while n >= x do
  begin
    print (x * x);
    set x := x + 1
  end
end
```

For excitement we need to add **functions/procedures** to the language!