

Compiler Construction Lent Term 2014

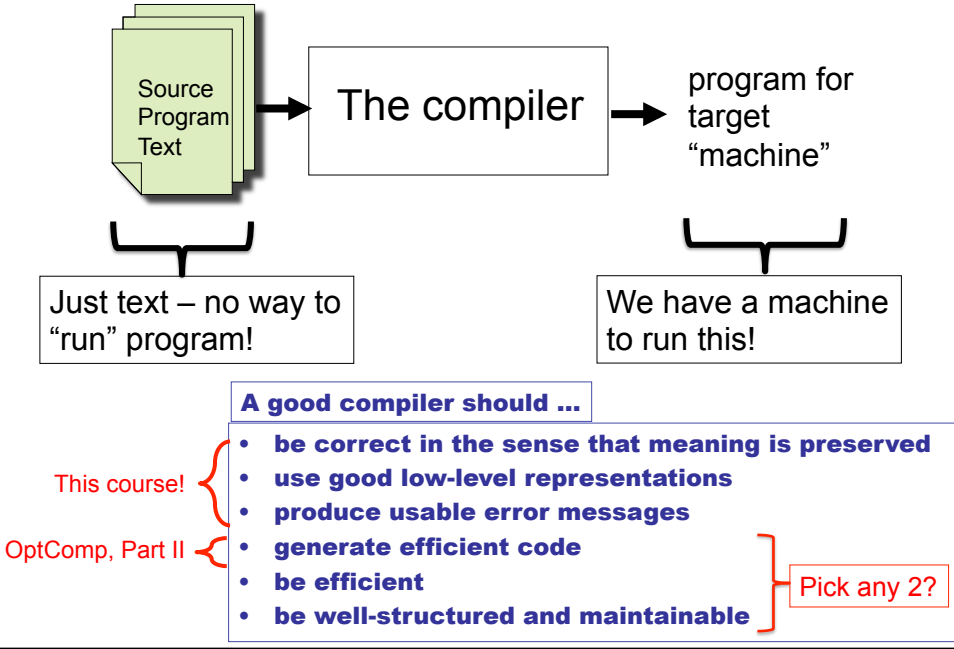
Lectures 1 - 4 (of 16)

Timothy G. Griffin
tgg22@cam.ac.uk

Computer Laboratory
University of Cambridge

1

Compilation is a special kind of translation



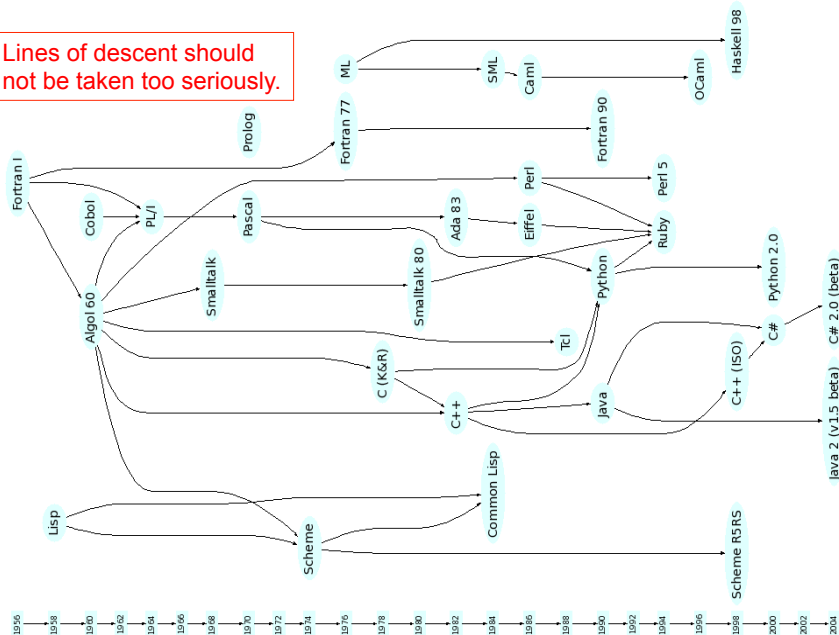
Why Study Compilers?

- 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.
- Compilers are a Computer Science success story illustrating the hallmarks of our field --- higher-level abstractions implemented with lower-level abstractions.
- **Every Computer Scientist should have a basic understanding of how compilers work.**

3

New languages will continue to evolve ...
From <http://merd.sourceforge.net/pixel/language-study/diagram.html>

Lines of descent should not be taken too seriously.



4

Mind The Gap

High Level Language

- Machine independent
- Complex syntax
- Complex type system
- Variables
- Nested scope
- Procedures, functions
- Objects
- Modules
- ...

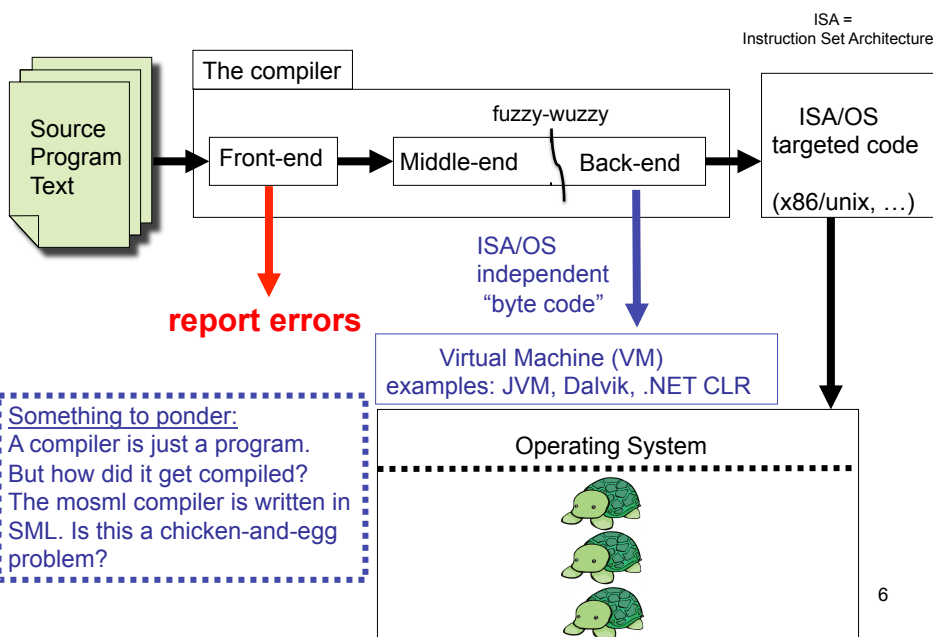
Typical Target Language

- Machine specific
- Simple syntax
- Simple types
- memory, registers, words
- Single flat scope

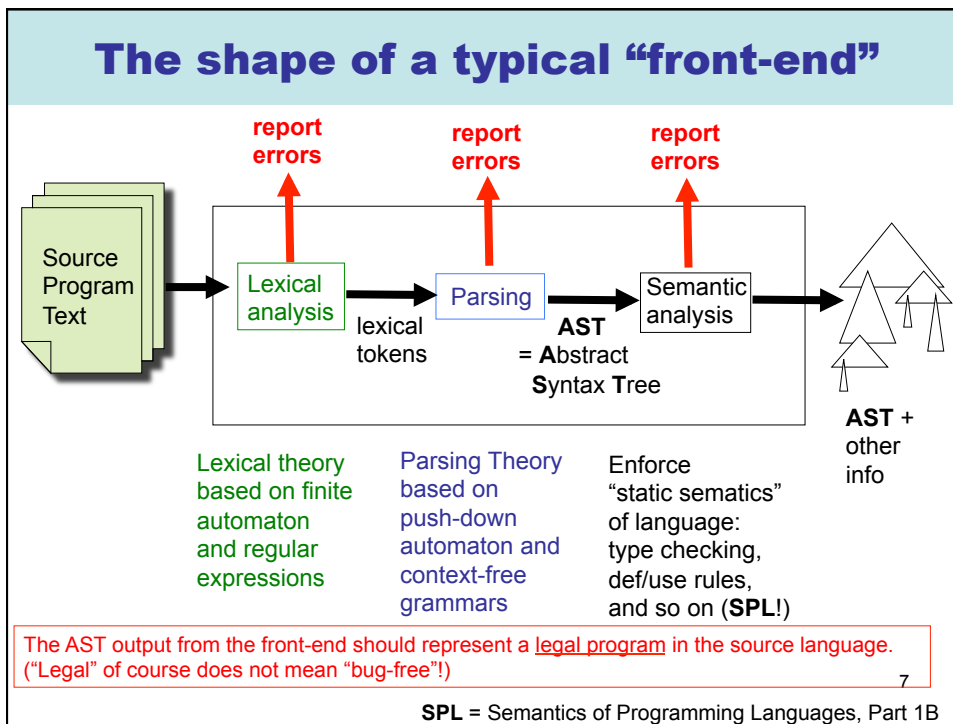
Help!!! Where do we begin???

5

Conceptual view of a typical compiler

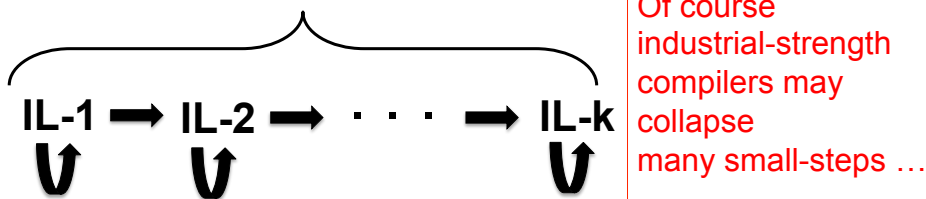


The shape of a typical “front-end”



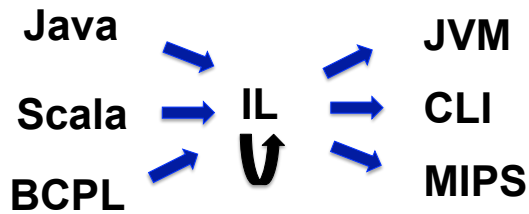
Our view of the middle- and back-ends : a sequence of small transformations

Intermediate Languages



- Each **IL** has its own semantics (perhaps informal)
- Each transformation (→) preserves semantics (**SPL!**)
- Each transformation eliminates only a few aspects of **the gap**
- Each transformation is fairly easy to understand
- Some transformations can be described as “optimizations”
- In principle (but not in practice), each **IL** could be associated with its own “machine” (so the line between “interpreter” and “machine” is fuzzy-wuzzy).

Another view



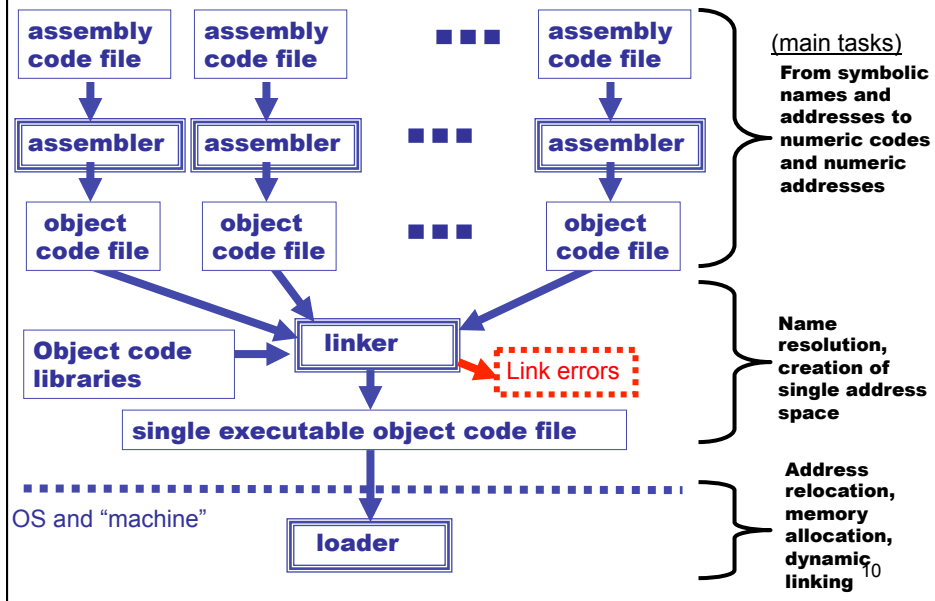
- One **IL** to rule them all
- Difficult to derive an **IL** if one has never seen a compiler before
- For instructional purposes we prefer to introduce multiple **ILs**

Example : search for "LLVM IR"

9

Oh yes, Assembly, Linking, Loading ...

This functionality may or may not be implemented in "the compiler".



Simple language (Slang) compilers

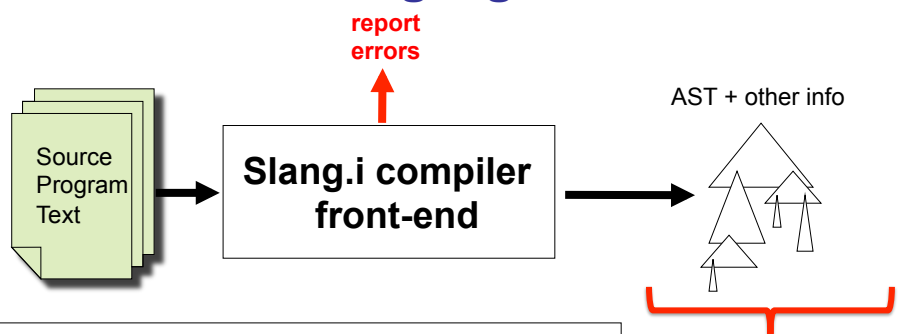
The lectures will center around *compiler concepts*, mostly illustrated by developing Slang compilers.

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

11

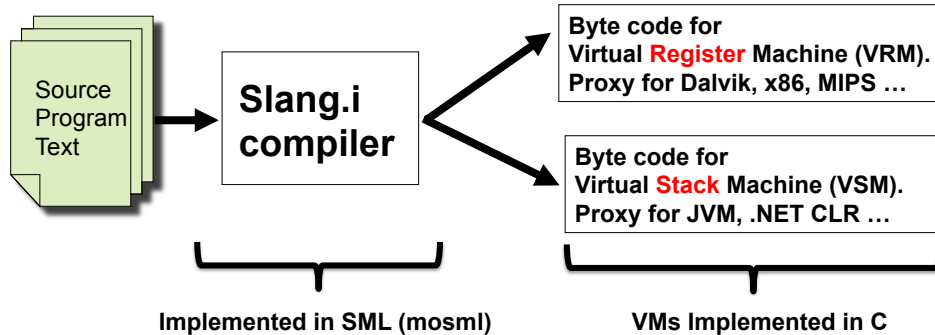
Slang is (bad?) concrete syntax for SPL languages



- **Why use L3+Objects?**
- **Why define yet another toy language?**
- **SPL gives us clear type system**
- **SPL gives us clear semantics**
- **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
Slang.4 VRM.2 and VSM.2	10.	Memory Management ("garbage collection")
	11.	Assorted topics : Bootstrapping, Exceptions
	12.	Objects (delayed to ensure coverage in SPL)
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

- Main text** • **Course Notes (by Prof Alan Mycroft and his predecessors).**
- Main textbook(s)** • **Compiler Design in Java/C/ML (3 books). Appel. (1996)**
- **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)**

15

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 !!!

16

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 * expr
| Deref of loc
| Seq of expr * expr
| If of expr * expr * expr
| While of expr * expr
| Print of expr
  
```

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

```

% 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)
  
```

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

19

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 L-values may be determined at run-time:

A[j*2] = j + 3;

20

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".

21

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/>

22

A peek into slang1/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.

23

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
```

24

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)

25

On to Context Free Grammars (CFGs)

$E ::= ID$

$E ::= NUM$

$E ::= E * E$

$E ::= E / E$

$E ::= E + E$

$E ::= E - E$

$E ::= (E)$

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

$E ::= ID \mid NUM \mid E * E \mid E / E \mid E + E \mid E - E \mid (E)$

26

CFG Derivations

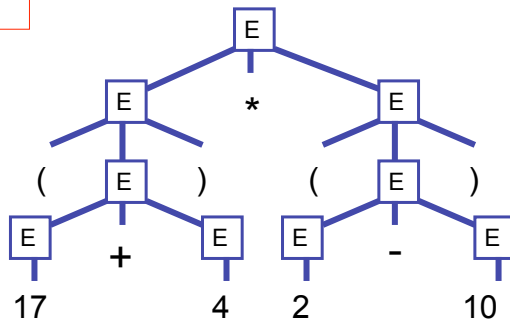
(G1) $E ::= ID \mid NUM \mid ID \mid E * E \mid E / E \mid E + E \mid E - E \mid (E)$

$E \rightarrow E * E$
 $\rightarrow E * (E)$
 $\rightarrow E * (E - E)$
 $\rightarrow E * (E - 10)$
 $\rightarrow E * (2 - 10)$
 $\rightarrow (E) * (2 - 10)$
 $\rightarrow (E + E) * (2 - 10)$
 $\rightarrow (E + 4) * (2 - E)$
 $\rightarrow (17 + 4) * (2 - 10)$

Rightmost derivation

$E \rightarrow E * E$
 $\rightarrow (E) * E$
 $\rightarrow (E + E) * E$
 $\rightarrow (17 + E) * E$
 $\rightarrow (17 + 4) * E$
 $\rightarrow (17 + 4) * (E)$
 $\rightarrow (17 + 4) * (E - E)$
 $\rightarrow (17 + 4) * (2 - E)$
 $\rightarrow (17 + 4) * (2 - 10)$

Leftmost derivation



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

27

More formally, ...

- **A CFG 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 $S ::= (S) \mid SS \mid$

28

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 β can be derived from α in 0 or more single steps
- $\alpha \Rightarrow^+ \beta$ means that β can be derived from α in 1 or more single steps

29

$L(G)$ = The Language Generated by Grammar G

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

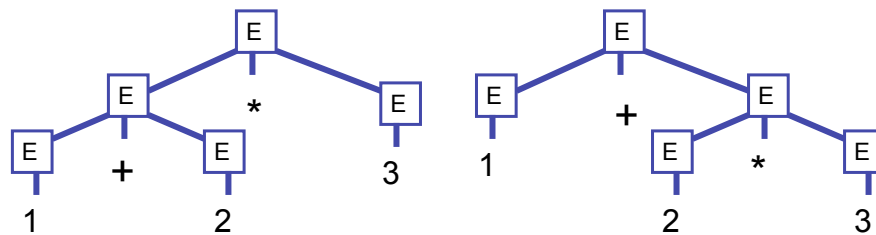
$$L(G) = \{w \in T^* \mid S \Rightarrow^+ w\}$$

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

30

Ambiguity

(G1) $E ::= ID \mid NUM \mid ID \mid E * E \mid E / E \mid E + E \mid E - E \mid (E)$



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!

31

Problem: Generation vs. Parsing

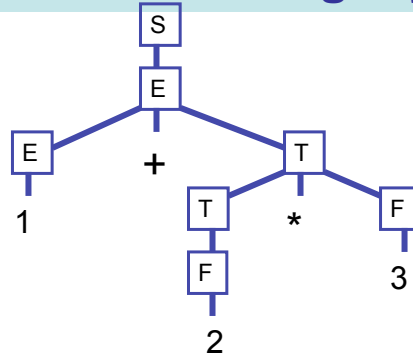
- **Context-Free Grammars (CFGs) describe how to to *generate***
- ***Parsing* is the inverse of generation,**
 - Given an input string, is it in the language generated by a CFG?
 - If so, construct a derivation tree (normally called a *parse tree*).
 - Ambiguity is a big problem

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 ...

32

We can often modify the grammar in order to eliminate ambiguity

(G2)
 $S ::= E\$$ (start, \$ = EOF)
 $E ::= E + T$ (expressions)
 | $E - T$
 | T
 $T ::= T * F$ (terms)
 | T / F
 | F
 $F ::= \text{NUM}$ (factors)
 | ID
 | (E)



This is the unique derivation tree for the string

1 + 2 * 3\$

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

33

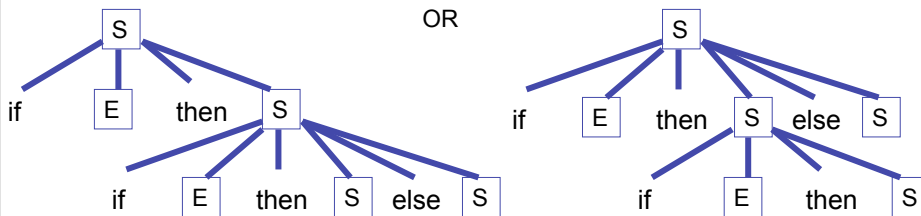
Famously Ambiguous

(G3) $S ::= \text{if } E \text{ then } S \text{ else } S \mid \text{if } E \text{ then } S \mid \text{blah-blah}$

What does

if e1 then if e2 then s1 else s3

mean?



34

Rewrite?

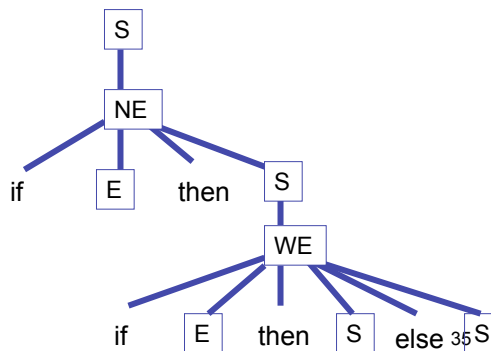
(G4)
 $S ::= WE \mid NE$
 $WE ::= \text{if } E \text{ then } WE \text{ else } WE \mid \text{blah-blah}$
 $NE ::= \text{if } E \text{ then } S$
 $\quad \mid \text{if } E \text{ then } WE \text{ else } NE$

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!

36

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 ($E ::= E + T \mid E - T \mid T$) will cause
code based on this method to go into an infinite loop!

37

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?

38

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

39

A peek at slang/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, 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
```

40

Types : SPL give us the rules

$$\begin{array}{ll}
 \text{(int)} \quad \Gamma \vdash n:\text{int} \quad \text{for } n \in \mathbb{Z} & \text{(deref)} \quad \frac{\Gamma(\ell) = \text{intref}}{\Gamma \vdash !\ell:\text{int}} \\
 \text{(bool)} \quad \Gamma \vdash b:\text{bool} \quad \text{for } b \in \{\text{true}, \text{false}\} & \text{(skip)} \quad \Gamma \vdash \text{skip}:\text{unit} \\
 \text{(op +)} \quad \frac{\Gamma \vdash e_1:\text{int} \quad \Gamma \vdash e_2:\text{int}}{\Gamma \vdash e_1 + e_2:\text{int}} & \text{(op } \geq) \quad \frac{\Gamma \vdash e_1:\text{int} \quad \Gamma \vdash e_2:\text{int}}{\Gamma \vdash e_1 \geq e_2:\text{bool}} \\
 \text{(if)} \quad \frac{\Gamma \vdash e_1:\text{bool} \quad \Gamma \vdash e_2:T \quad \Gamma \vdash e_3:T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3:T} & \text{(seq)} \quad \frac{\Gamma \vdash e_1:\text{unit} \quad \Gamma \vdash e_2:T}{\Gamma \vdash e_1; e_2:T} \\
 \text{(assign)} \quad \frac{\Gamma(\ell) = \text{intref} \quad \Gamma \vdash e:\text{int}}{\Gamma \vdash \ell := e:\text{unit}} & \text{(while)} \quad \frac{\Gamma \vdash e_1:\text{bool} \quad \Gamma \vdash e_2:\text{unit}}{\Gamma \vdash \text{while } e_1 \text{ do } e_2:\text{unit}}
 \end{array}$$

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

SPL give us some options ...

Language design 3. Store initialization

Recall that

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

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

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

Instead, could

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

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

We like the first option!

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
...
```

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

Theorem: if

$(t, e') = \text{css } e$

Then

$\vdash e : t$

and $\text{erase}(e') = e$,
where erase removes
all type annotations.

Prove by induction on the
structure of e .

Not interesting in Slang.1,
but later

43

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 types

(In later versions of
Slang we will do interesting
type annotations here....)

```
n := 10;
x := 1;
while (!n >= !x) do
(
  print(!x * !x);
  x := !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**
- **For Slang.1 the L1 semantics keeps us more-or-less honest**
- **Computation in registers requires arguments to have a location**
- **Computation at the “top of the stack” allows arguments to be implicit**

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 ...

45

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 a register-oriented VM of Android**

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

46

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:

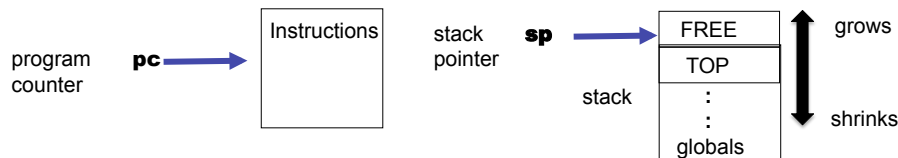
No more than 256 instructions per program

"Only" 256 registers

Only 13 basic instructions

49

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.

50

Mind the Gap --- Three main issues ---

AST_L1 (output of front-end)

Low-level code

1 No (syntactic) distinction between expressions and side-effecting statements

Manipulating values and state are very distinct

2 Structured control operations, **If-then-else, while-do**

Flat sequence of operations, control via jumps and labels

3 “Unnamed” sub-expression (one of FORTRAN’s major innovations!)

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

```
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” locations (registers, memory)

Not an issue with stack-oriented machines (next slide)

(code not optimal!)

Stack machine does not need to “name” subexpressions

$e1 \text{ op } e2$



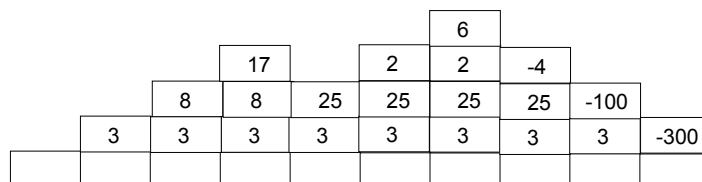
... code for e1...

... code for e2 ...

op

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

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



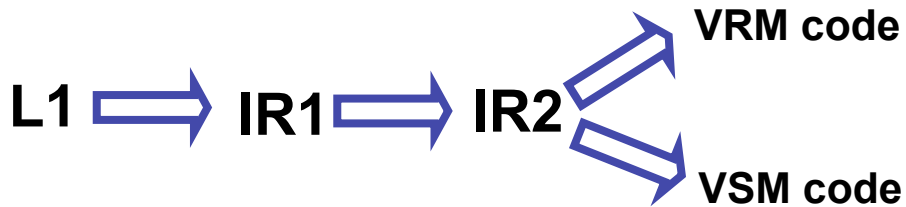
Bridging the Gap with Intermediate Languages

IR1 (Intermediate Language 1)

Make distinction between expressions
and side-effecting statements

IR2 (Intermediate Language 2)

No structured control --- just sequence
of instructions



53

from slang1/compile.sml

```
fun back_end fout ast =
  case !target of
    VSM => emit_vsm_bytecode fout
           (vsm_assemble
            (vsm_code_gen ast))
  | VRM => emit_vrm_bytecode fout
           (vrm_assemble
            (vrm_code_gen ast))

fun compile fin fout =
  back_end fout
  (translate_ir1_to_ir2
   (translate_l1_to_ir1
    (check_types
     (parse (init_lex_buffer fin))))))
```

AST for IR1

AST_L1.sml

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

AST_IR1.sml

```
type unary_oper = AST_L1.unary_oper  
type oper       = AST_L1.oper  
type loc        = AST_L1.loc  
  
datatype ir1_expr =  
  IR1_ESkip  
| IR1_Integer of int  
| IR1_Boolean of bool  
| IR1_UnaryOp of unary_oper * ir1_expr  
| IR1_Op of ir1_expr * oper * ir1_expr  
| IR1_Deref of loc  
| IR1_EIf of ir1_expr * ir1_expr * ir1_expr  
  
datatype ir1_stm =  
  IR1_Expr of ir1_expr  
| IR1_Assign of loc * ir1_expr  
| IR1_Seq of ir1_stm list  
| IR1_SIf of ir1_expr * ir1_stm * ir1_stm  
| IR1_While of ir1_expr * ir1_stm  
| IR1_Print of ir1_expr
```

Notice that expressions of type `ir1_expr` are now free of side-effects.

55

from L1 to IR1

```
translate_l1_to_ir1 : expr -> ir1_stm
```

```
fun translate_l1_to_ir1 e =  
  let val (sl, e') = l1_to_ir2 e  
  in  
    IR1_Seq (sl @ [IR1_Expr e'])  
  end
```

Here's the idea. If we have

```
(sl, e') = l1_to_ir2 e
```

and `e` evaluates to value `V`, then “running” `sl` and then evaluating `e'` will also result in value `V`.

l1_to_ir2 : the easy bits

```
fun l1_to_ir2 Skip = ([], IR1_ESkip)
| l1_to_ir2 (Integer n) = ([], IR1_Integer n)
| l1_to_ir2 (Boolean b) = ([], IR1_Boolean b)
| l1_to_ir2 (UnaryOp (uop, e)) =
  let val (sl, e') = l1_to_ir2 e in (sl, IR1_UnaryOp(uop, e')) end
| l1_to_ir2 (Assign (l, e)) =
  let val (sl, e') = l1_to_ir2 e in (sl @ [IR1_Assign(l, e')], IR1_ESkip) end
| l1_to_ir2 (Deref l) = ([], IR1_Deref l)
| l1_to_ir2 (Seq (e1, e2)) =
  let val (sl1, _) = l1_to_ir2 e1
      and (sl2, e2') = l1_to_ir2 e2
  in
    (sl1 @ sl2, e2')
  end
| l1_to_ir2 (If(e1, e2, e3)) =
  let val (sl1, e1') = l1_to_ir2 e1
      and (sl2, e2') = l1_to_ir2 e2
      and (sl3, e3') = l1_to_ir2 e3
  in
    (* would be better to avoid duplication of e1'? *)
    (sl1 @ [IR1_SIf(e1', IR1_Seq sl2, IR1_Seq sl3)], IR1_EIf(e1', e2', e3'))
  end
| l1_to_ir2 (Print e) =
  let val (sl, e') = l1_to_ir2 e in (sl @ [IR1_Print e'], IR1_ESkip) end
... ..
... ..
```

Oh no --- a tricky bit

```
... ..
| l1_to_ir2 (Op (e1, bop, e2)) =
  let val (sl1, e1') = l1_to_ir2 e1
      and (sl2, e2') = l1_to_ir2 e2
  in
    (sl1 @ sl2, IR1_Op(e1', bop, e2'))
  end
... ..
```

Correct?

No!

slang1/examples/nested.slang

```
% 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
```

Counter example:

Problem : running sl2 could change the locations read by e1'

One solution for tricky bit

```

... ""
| l1_to_ir2 (Op (e1, bop, e2)) =
  let val (s11, e1') = l1_to_ir2 e1
      and (s12, e2') = l1_to_ir2 e2
  in
    if Library.intersects(read_locations_of e1, write_locations_of e2)
    then let val l = Global.new_loc ()
        in
          (s11 @ [IR1_Assign(l, e1')] @ s12, IR1_Op(IR1_Deref l, bop, e2'))
        end
      else (s11 @ s12, IR1_Op(e1', bop, e2'))
    end
  end
... ""

```

Similar problem with while-loop, but simpler solution:

```

... ""
| l1_to_ir2 (While (e1, e2)) =
  let val (s11, e1') = l1_to_ir2 e1
      and (s12, _ ) = l1_to_ir2 e2
  in
    (s11 @ [IR1_While(e1', IR1_Seq (s12 @ s11))], IR1_ESkip)
  end
... ""

```

AST for IR2 (flat sequence of statements)

AST_IR1.sml

```

type unary_oper = AST_L1.unary_oper
type oper       = AST_L1.oper
type loc        = AST_L1.loc

datatype ir1_expr =
  IR1_ESkip
  | IR1_Integer of int
  | IR1_Boolean of bool
  | IR1_UnaryOp of unary_oper * ir1_expr
  | IR1_Op of ir1_expr * oper * ir1_expr
  | IR1_Deref of loc
  | IR1_EIf of ir1_expr * ir1_expr * ir1_expr

datatype ir1_stm =
  IR1_Expr of ir1_expr
  | IR1_Assign of loc * ir1_expr
  | IR1_Seq of ir1_stm list
  | IR1_Sif of ir1_expr * ir1_stm * ir1_stm
  | IR1_While of ir1_expr * ir1_stm
  | IR1_Print of ir1_expr

```

AST_IR2.sml

```

type loc = AST_L1.loc
type label = string

type ir2_expr = AST_IR1.ir1_expr

datatype ir2_stm =
  IR2_Label of label
  | IR2_Expr of ir2_expr
  | IR2_Assign of loc * ir2_expr
  | IR2_Jump of label
  | IR2_Fjump of ir2_expr * label
  | IR2_Print of ir2_expr

type program = ir2_stm list

```

IR2_Jump l : unconditional jump to code after label l
 IR2_Fjump(e, l) : jump to l only if e evaluates to 0

60

IR1 to IR2

```
fun translate_ir1_to_ir2 stm = (flatten stm) @ [IR2_Halt]
```

It is easy to flatten a sequence :

e_1; e_2; ... e_n;



... code for e_1...

**:
:**

... code for e_n ...

```
| flatten (IR1_Seq sl) = List.concat (List.map flatten sl)
```

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

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

Remember these?

Flatten conditional, while loop

if e1 then e2 else e3



```
IR2_Fjump(e1, k)  
... code for e2 ...  
IR2_Jump m  
IR2_Label k  
... code for e3 ...  
IR2_Label m
```

while e1 do e2



```
IR2_Label k  
IR2_Fjump(e1, m)  
... code for e2 ...  
IR2_Jump k  
IR2_Label m
```

From slang1/IR1_to_IR2.sml

```
fun flatten (IR1_Expr e) = [IR2_Expr e]
| flatten (IR1_Assign(l, e)) = [IR2_Assign(l, e)]
| flatten (IR1_Seq sl) = List.concat (List.map flatten sl)
| flatten (IR1_SIf(e, stm1, stm2)) =
  let val sl1 = flatten stm1
      and sl2 = flatten stm2
      and else_label = Global.new_label ()
      and end_label = Global.new_label ()
  in
    (IR2_Fjump (e, else_label)) ::
    (sl1 @
     ((IR2_Jump end_label) ::
      ((IR2_Label else_label) ::
       (sl2 @
        [IR2_Label end_label]))))
  end
| flatten (IR1_While(e, stm)) =
  let val sl = flatten stm
      and start_label = Global.new_label ()
      and end_label = Global.new_label ()
  in
    (IR2_Label start_label) ::
    ((IR2_Fjump (e, end_label)) ::
     (sl @
      [IR2_Jump start_label,
       IR2_Label end_label]))
  end
| flatten (IR1_Print e) = [IR2_Print e]
```

63

From slang1/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 =
  VRM_comment of vrm_comment
| VRM_Label of vrm_code_loc
(* data operations *)
| VRM_Nop
| VRM_Set of vrm_data_loc * vrm_constant
| VRM_Mov of vrm_data_loc * vrm_data_loc
| VRM_Add of vrm_data_loc * vrm_data_loc * vrm_data_loc
| VRM_Sub of vrm_data_loc * vrm_data_loc * vrm_data_loc
| VRM_Mul of vrm_data_loc * vrm_data_loc * vrm_data_loc
(* control flow operations *)
| VRM_Hlt
| VRM_Jmp of vrm_code_loc
| VRM_Ifz of vrm_data_loc * vrm_code_loc
| VRM_Ifp of vrm_data_loc * vrm_code_loc
| VRM_Ifn of vrm_data_loc * vrm_code_loc
(* input/output *)
| VRM_Pri of vrm_data_loc

type vrm_assembler = vrm_operation list
```

64

command-line examples

- | | |
|--|---|
| <ul style="list-style-type: none"> • 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 | <ul style="list-style-type: none"> • vrmo examples/squares.vrmo
run VRM.0 on bytecode file • vrmo -v examples/squares.vrmo
same as above, but with verbose output • vrmo -s examples/squares.vrmo
just print the bytecode |
| <ul style="list-style-type: none"> • 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 | <ul style="list-style-type: none"> • vsmo examples/squares.vsmo
run VSM.0 on bytecode file • vsmo -v examples/squares.vsmo
same as above, but with verbose output • vsmo -s examples/squares.vsmo
just print the bytecode |

65

Slang.1 example (squares.slang)

<pre>begin set n := 10; set x := 1; while n >= x do begin print (x * x); set x := x + 1 end end</pre>	<pre>n := 10; x := 1; while (!n >= !x) do (print(!x * !x); x := !x + 1)</pre>	<pre>n := 10; x := 1; _LO : Fjump (!n >= !x) _L1 : print((!x * !x)); x := (!x + 1); Jump _LO; _L1 : skip; Halt</pre>
--	--	---

Parse, type check, L1 to L2

IR1 to IR2

VSM path

IR2

```

n := 10;
x := 1;
_LO : Fjump (!n >= !x)
_L1 : print(!x * !x);
      x := (!x + 1);
      Jump _LO;
_L1 : skip;
      Halt
    
```

IR1 to IR2

```

push 0      %slot for n
push 0      %slot for x
push 10
store 0     %store n
push 1
store 1     %store x
_LO : load 0 %start >= ... load n
      load 1 %load x
      sub
      ifn _L2
      push 1 %push true
      jmp _L3
_L2 : push 0 %push false
_L3 : ifz _L1 % ... end >=
      load 1 %load x
      load 1 %load x
      mul
      pri
      load 1 %load x
      push 1
      add
      store 1 %store x
      jmp _LO
_L1 : push 0
      hlt      % that's all folks!
    
```

VRM path

IR2

```

n := 10;
x := 1;
_LO : Fjump (!n >= !x)
_L1 : print(!x * !x);
      x := (!x + 1);
      Jump _LO;
_L1 : skip;
      Halt
    
```

IR1 to IR2

```

set _Zero 0      %used for zero, false, and Eskip
set _One 1      %used for one and true constants
set _TEMP_0 10
mov n _TEMP_0
set _TEMP_1 1
mov x _TEMP_1
_LO : sub _TEMP_3 n x %start >= ...
      ifn _TEMP_3 _L2
      set _TEMP_2 1
      jmp _L3
_L2 : set _TEMP_2 0
_L3 : ifz _TEMP_2 _L1 %end >=
      mul _TEMP_4 x x
      pri _TEMP_4
      set _TEMP_5 1
      add _TEMP_6 x _TEMP_5
      mov x _TEMP_6
      jmp _LO
_L1 : hlt      % that's all folks!
    
```

Clearly could do better! See Problem Set 1!

VRM assemble (VSM assemble not shown)

```
set_Zero 0
set_One 1
set_TEMP_0 10
mov n_TEMP_0
set_TEMP_1 1
mov x_TEMP_1
_L0: sub_TEMP_3 n x
ifn_TEMP_3_L2
set_TEMP_2 1
jmp_L3
_L2: set_TEMP_2 0
_L3: ifz_TEMP_2_L1
mul_TEMP_4 x x
pri_TEMP_4
set_TEMP_5 1
add_TEMP_6 x_TEMP_5
mov x_TEMP_6
jmp_L0
_L1: hlt
```

assemble

```
l0 : set r0 0
l1 : set r1 1
l2 : set r2 10
l3 : mov r3 r2
l4 : set r4 1
l5 : mov r5 r4
l6 : sub r6 r3 r5
l7 : ifn r6 l10
l8 : set r7 1
l9 : jmp l11
l10 : set r7 0
l11 : ifz r7 l18
l12 : mul r8 r5 r5
l13 : pri r8
l14 : set r9 1
l15 : add r10 r5 r9
l16 : mov r5 r10
l17 : jmp l6
l18 : hlt
```

Replace symbolic labels and locations with numeric values

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
```

70

An example where L1 to IR1 is actually interesting

slang1/examples/nested.slang

```
% 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
```

Translates to this
IR2 program

```
x := 10;  
x := (4 * !x);  
_X0 := !x;  
x := (2 * !x);  
x := (!_X0 - !x);  
print(!x);  
skip;  
halt
```

Hmmmm, I wonder what that **skip** is doing there