# dOvs Eksamens Noter

Hugh Benjamin Zachariae, Mac, Flamingo

January 2020

## Contents

# 1 Compiler intro

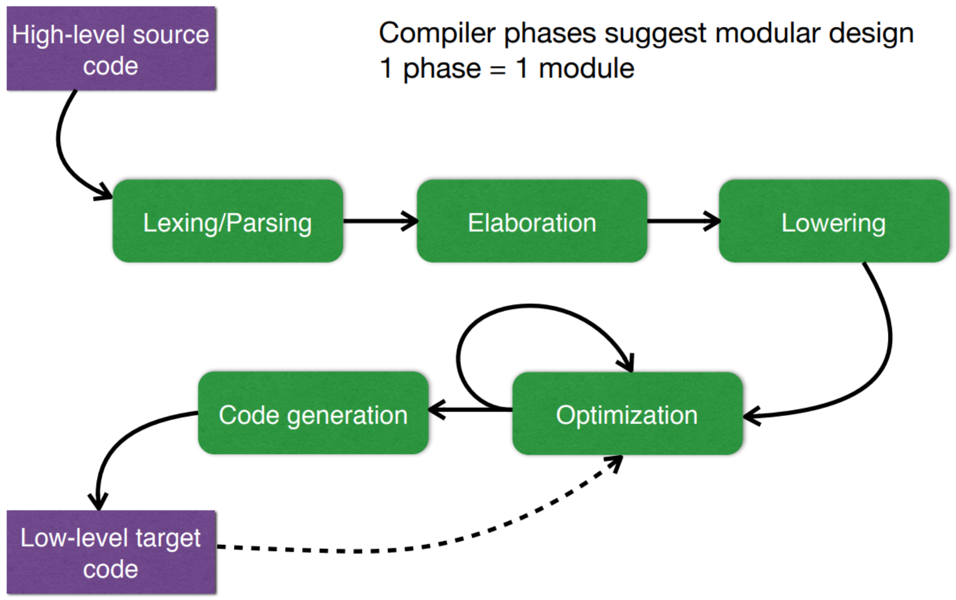


Figure 1.1: Compiler modular phases.

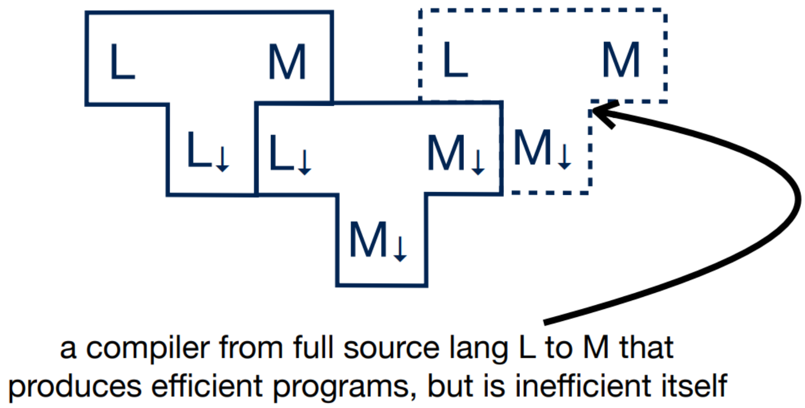


Figure 1.2: Bootstrap compiling

- **Lexing/Parsing**: String $\rightarrow_{lexing}$ Tokens $\rightarrow_{parsing}$ Abstract Syntax Tree (AST)
- **Elaboration**: *Resolving scope* and *Type checking*. Most errors found here.
- **Lowering**: High-level features to target-language like constructs (e.g. assembly-like). *Intermediate representation*, LLVM.
- **Optimization**: Detect and rewrite expensive operations. Lifting invariants out of loops, parallelization.
- **Code generation**: fx LLVM to X86 (registers, instruction etc.)

- **Bootstrapping compilers**: Compile your language in your own language.

# 2 Lexical

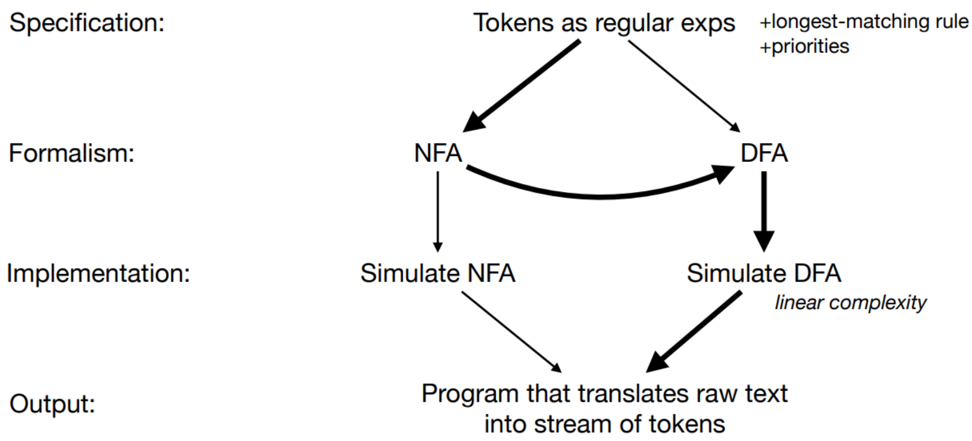


Figure 2.1: REG to NFA to DFA

- Tokens: E.g. ID("a"), INT, IF etc. Some tokens include metadata like names in ID.
- Non-tokens: comments, whitespace etc.
- REG → NFA → (closures) DFA → Minimized DFA (more effective)
- REG: Handle priorities and longest matching string token wins.
- Ocamllex: Lexer generator

# 3 Parsing

A context-free grammar (CFG) is a 4-tuple $G = (V, \Sigma, S, P)$

- V is a finite set of *nonterminal* symbols
- $\Sigma$ is an alphabet of *terminal* symbols and $V \cap \Sigma = \emptyset$
- $S \in V$ is a *start* symbol
- P is a finite set of *productions* of the form $A \rightarrow \alpha$, where
  - $A \in V$, i.e., A is a nonterminal, and
  - $\alpha \in (V \cup \Sigma)^*$, i.e., $\alpha$ is possibly empty string of nonterminals or terminals

Figure 3.1: CFG Definition

$S \rightarrow$ if E then S else S
$S \rightarrow$ begin S L
$S \rightarrow$ print E
$L \rightarrow$ end
$L \rightarrow$ ; S L
$E \rightarrow$ num = num

- FIRST ($\alpha$) : set of terminals that begin strings derived from $\alpha$
- FOLLOW(X): set of terminals a that can appear immediately to the right of X in some derivable string, e.g., $S \Rightarrow^* \alpha X a \beta$
- Let nullable(X) be true when X can derive empty string $\varepsilon$

| Nonterminal | Nullable? | First set | Follow set |
|---|---|---|---|
| **S** | | if, begin, print | else, end, ;, $ |
| **L** | | end, ; | else, end, ;, $ |
| **E** | | num | then, else, end, ; $ |

Figure 3.2: Top-down parsing table. You do not want more than one possibility in a cell.

- Abstract Syntax Tree (AST):
- Context-Free Grammars (CFG):
  - *Terminals* $\rightarrow$ *production rules*
  - Terminals are leafs in the tree (e.g. x, y).
  - Non-Terminals are links in the tree (e.g. BinExp)
  - Definition see figure 3.1.
  - Ambiguity: You don't want ambiguity, you want determinism. *Associativity* (right/left) and *precedence* (e.g. times before plus).
- Top-down/Bottom-up parsing:
  - Top-down is predictive parsing:

    - leftmost derivation
    - "see whats coming"
    - Breaks down at for example: $S \to S + x \mid S - x \mid x$. Here you don't know what to do when you see an $x \ldots$
    - See figure 3.2 for parsing table.
  - Bottom-up: **LR parsing** is rightmost reduction.
    - Rightmost reduction
    - Includes EOF "$" symbol.

## 3.1 LR parsing

**Bottom-up**:

- Rightmost reduction
- Includes EOF "$" symbol.

**Terms**:

- An **Item** is a hypothesis about sub-derivations: $N$ is hypothesis, $\alpha$ is confirmed to be parsed, $\beta$ is to be confirmed, $N \to \alpha.\beta$. Notice that it looks like a production rule, but with a dot somewhere in it.
- Item is reducible if $\beta$ is empty. The right side of the dot is empty.
- $\epsilon$-closure of an item set: add new hypothesis to set if expecting a non-terminal. Accessible steps while doing lambda steps.
- Stack based: stack of alternating items sets and derivation trees.
- Conflicts: shift/reduce, reduce/reduce. You don't know what to do from one state, when seeing an input symbol.

**Operations**: Look up stack state, and input symbol to get action

- Reduce $k$: Pop stack as many times as the number of symbols on the right-hand side of rule $k$. Choose a grammar rule $X \to A\ B\ C$; pop $C, B, A$ from the top of the stack, and push X onto the stack. If dot is found on the right side of all symbols.
- Shift: Advance input one token; push token to stack. Go from one state to another after seeing a terminal input. Move dot one spot.
- Goto: Add hypethesis to stack - which sub-derivations we can go to. Goto state (move across edge). Go from one state to another after seeing a non-terminal. Move dot one spot.

Goto and shift must preserve the structure of the stack (item set > derivation).

**Examples**: All LR parsing examples

You can create a DFA by calculating first, the starting state and its closure. Then calculate the closures (dot in front of non-terminal) developed by shifting each terminal and non-terminal from that state (moving the dot after the shifted input symbol). Afterwards, you can develop a parsing table, *state* by terminal/non-terminal. See figure 3.3 for parsing table, DFA for shift reduce grammar.

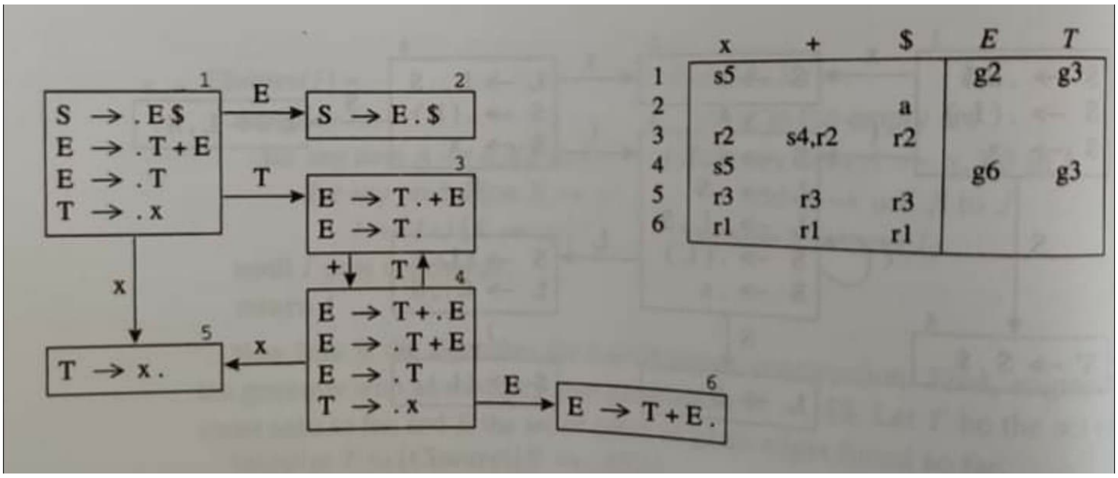


Figure 3.3: LR(0) shift/reduce conflict, parsing table and state DFA, s$n$: shift to state $n$

Reduction based on $k$ lookahead. The higher $k$, the less conflicts. However, more than 1 is not used for compilation, as the parsing table would be huge.

Since LR(0) needs no lookahead, we require one action for each state. With shift and reduce, we get a shift/reduce conflict.

LR(1) items consists of a *grammar production*, a *right-hand-side position* and a *lookahead symbol*. Choose whether or not to reduce based on stack and one lookahead on input.

Lookaheads are calculated by: Any state that contains an item of the form $A \rightarrow x.By\ \{t\}$, where $x$ and $y$ are arbitrary strings of terminals and nonterminals and $B$ is a nonterminal, you add an item of the form $B \rightarrow .w\ \{s\}$ for every production $B \rightarrow w$ and for every terminal in the set $s = FIRST(yt)$.

## 3.2 Scoping rules

Rules of programming language to regulate how names and ID's are resolved.

**Problems**:

- Nesting: Same name for variable in nested scopes. What value should we return?
- Forward reference: Using something before it is declared. E.g. mutual recursion.

**Scoping terms**:

- Scope of declaration: Part of the program where the declaration can be referred to.
- **Static nested scopes (SML style)**: Identifier scope is the smallest block (begin/end, function, or procedure body) containing the identifier's declaration. This means that an identifier declared in some block is only accessible within that block and from procedures declared within it.
  - Nearest visible: Return value of nearest declaration in the code.
  - Stack-like behavior

    - JS function-level lexical scoping: Inner functions contain the scope of parent functions even if the parent function has returned.

- **Static scoping**: Inner functions can access identifiers in outer scope. Can be deduced in compile time (C).
- **Dynamic scoping**: A function $p$ which prints $x$. Two functions, $d1$ and $d2$,that declare $x$ as 1 and 2 and then calls $p$. $d1$ will print 1 and $d2$ will print 2. I.e. the scope depends on the call stack and chain of function calls.

#### 3.2.1 Namespace

: Different declaration identifiers can reside in different syntactic namespaces. E.g. in Tiger: *var/function* are in the same namespace, but *type* is in another.
**Tiger scoping and namespaces**:

- Global: base types (`int`, `string`) and built-in functions (e.g. `print`).
- `let`, `function`, and `record` introduce name declarations.
- Scoping follows SML, static, and lexical.
- `FunctionDec`/`TypeDec` introduce mutual visibility to each uninterrupted group of declarations (e.g. gives mutual recursion).

#### 3.2.2 Environments

Symbol tables mapping names (`var` names to types, types to type decls, and functions to function specifications).

Static, lexical scoping means that we need to update environments upon entering the scope and undo updates after leaving.

# 4 Semantic analysis

Checking that input program is well-typed, catching most errors. Generates typed AST. Strictly speaking, positions is no longer necessary for this part, but we would like to report errors with positions.

**Nominal type equivalence**: Two classes, although declared identically are not the same. Therefore, Arrays and Records in Tiger.light has a unique integer reference (`type unique = int ref`).

**Type environment**: Symbol map of name/identifier (`symbol` type in implementation) to type.

**Name type**: Necessary for mutual recursion in types (Records or arrays). Two traversals: Creating placeholder `name` types. Changing static link placeholder with correct type in second traversal.

```
type unique = int ref

 type ty =
       INT
     | STRING
     | RECORD of (Symbol.symbol * ty) list * unique
     | ARRAY of ty * unique
     | NIL
     | UNIT
     | NAME of Symbol.symbol * ty option ref
     | ERROR (* ambiguity exists due to type error *)
```

Figure 4.1: Tiger light types

**Recusive functions**: Two traversals: First, collect function signatures (return and parameter types etc.) and add to environment. Second, for each function check that body matches declared signature.

**Nil types**: Can be assigned to all record types. Need to be taken into account during conditions, declarations, and assignments etc.

# 5 LLVM Intermediate Representation

Using LLVM as a bridge from many high-level languages, to many low-level languages. Makes it so that we only have to make $n + m$ combinations from $n \times m$.

**Low-level aspects**:

- Convenient abstraction over architecture specific issues (we do not want to deal with.)
  - Infinite registers (we do not care about exact purpose of registers)
  - Direction of stack growth
  - Calling convention
  - Exact instruction set
- Exposes low-level aspects important for code gen:
  - Notion of functions (call stack)
  - Allocation of stack during function execution
  - Storing and loading pointers (stack and heap locations)
  - Instructions for arithmetics
  - Instructions for jumps and conditional jumps

### 5.0.1 LLVM features

:

- Intermediate assembly like
- Infinite number of registers - can only be assigned to once (single static assignment)
- Types
  - First-class types*:
    - Single value: **integer**, float, x86_mmx, **pointer**, vector, label
    - Aggregate types: **arrays**, **structures**
  - Function type and void type
- Identifiers
  - Global (@)
  - Local (%)
  - LLVM– only allows named identifiers
- 

**Bold are used by us.*

### 5.0.2 Structure of LLVM

- Program:
  - Global decls (list of globals, e.g. string literals)
  - Types: list of named types
  - Function decls
- Function declaration: Header (param, return type), Control Flow Graph (Entry basic block + labeled basic blocks)
- Control Flow Graph:
  - Basic block: List of instructions + terminator
  - Terminator: Return or branch instruction

**Conversion form Tiger to LLVM**:

- *Mutable values*: In LLVM we can only assign to a register once. Instead use `alloca` to allocate space on the stack and `store` the pointer. Then use `load` and `save` to read and update the variable.
- *Mutable values continued*: The number of locals is statically known and at compile time, traverse AST to create identifiers for each variable with pointer.
- When compiling a function: Emit a bunch of `alloca`s for all the locals in the function first.
- *Nested functions*: Allocate all locals in record structure. Access locals and parent scope via GEP. Add static link to parent function as first argument in the function and first element in the locals structure.
- *Hoisting*: All functions are at the same level. Add variable/identifier offset (depth) to AST so that we can use it for GEP.
- *Conditionals*: Reserve return location and then store Then/Else bodies return there.
- *Allocation*: `alloca`s need to happen in the first basic block, so that they are not repeated, for example during a loop, growing the stack unnecessarily.
- *Records*: Nested records are flattened into `i8*` (pointer) and accessed via GEP. Records are allocated on the heap. The size is calculated via runtime C (`calloc`).
- *Arrays*: Same as for records (allocation, size and static links).

## 5.1 Closure conversion

In functional programming a function passed as an argument is represented as a closure: Pointer and means of accessing non-local variables (free variables).

```
F = {fun*, env*}
```

All functions are top-level and closed.

Closure conversion phase in functional compiling transforms the functions so non of them appear to access free variables, turning all free variable access into formal parameter access.

$f(a_1, a_2, \ldots, a_n) = B$ at depth $d$
$x_1, x_2, \ldots, x_n$ are escaping local variables,
$y_1, y_2, \ldots, y_n$ are non-escaping local variables,
rewrite into:
$f(a_0, a_1, \ldots, a_n) = \texttt{let var } r := \{a_0, x_1, x_2, \ldots, x_n\} \texttt{ in } B' \texttt{ end}$

- New parameter $a_0$ is the static link.
- Variable $r$ is a record containing all escaping variables and the static link.
- The variable $r$ becomes the static link argument when calling a function at depth $d + 1$.
- In LLVM, use the GEP function to access free variables in some offset of $r$.

These are called *linked closures*. In immutable variable languages, just copy all closed values into closure instead of static link (*flat closures*). *Hybrid closures* are the best of both worlds.

### 5.1.1 First-class (Higher-order) functions

You can call a function, that has two arguments, with only one argument. It then return a function call variable, that you can call with the last argument. Closure conversion also hoists first-class functions into top-level and closed functions.