dOvs Eksamens Noter

Hugh Benjamin Zachariae

January 2020

Contents

1 Compiler intro
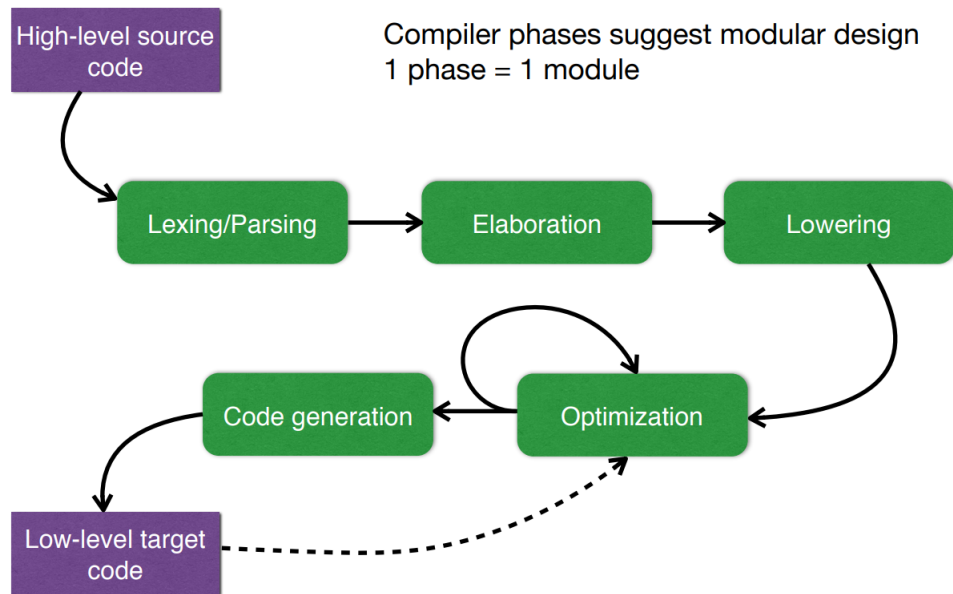
Figure 1: Compiler modular phases.

a compiler from full source lang L to M that
produces efficient programs, but is inefficient itself
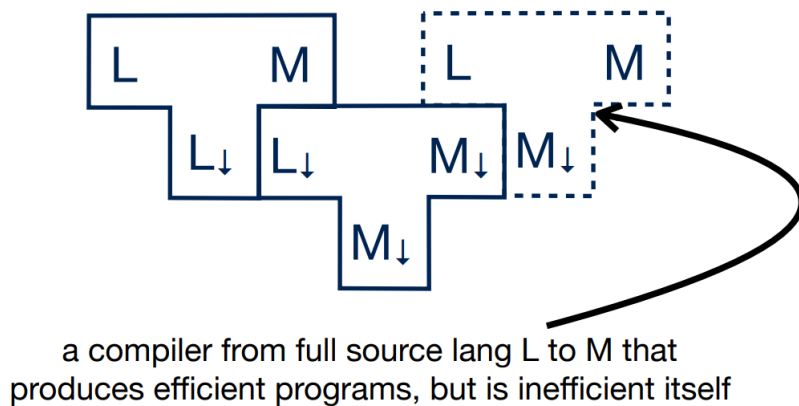
Figure 2: Bootstrap compiling

- **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.)

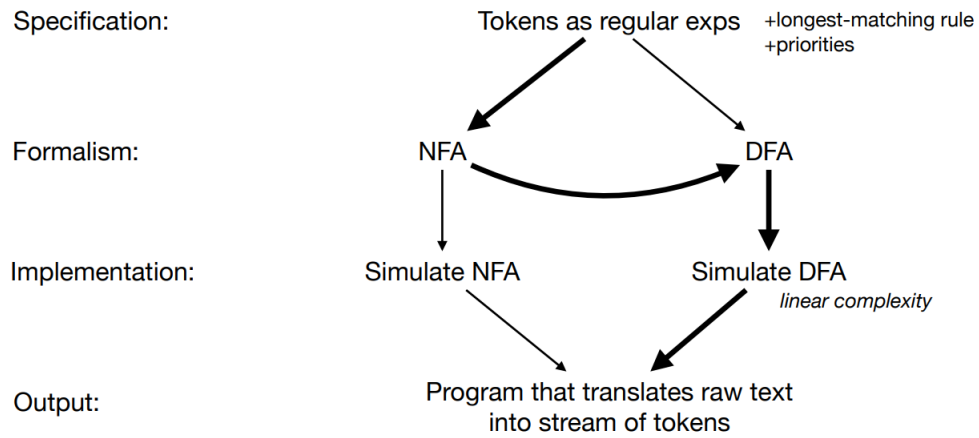- **Bootstrapping compilers**: Compile your language in your own language.

2 Lexical

Figure 3: 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, Σ, S, P)

- V is a finite set of *nonterminal* symbols
- Σ is an alphabet of *terminal* symbols and V ∩ Σ = ∅
- S ∈ V is a *start* symbol
- P is a finite set of *productions* of the form A → α, where
    - A ∈ V, i.e., A is a nonterminal, and
    - α ∈ (V ∪ Σ)*, i.e., α is possibly empty string of nonterminals or terminals

Figure 4: CFG Definition

S → if E then S else S
S → begin S L
S → print E
L → end
L → ; S L
E → num = num

- FIRST (α) : set of terminals that begin strings derived from α
- FOLLOW(X): set of terminals a that can appear immediately to the right of X in some derivable string, e.g., S ⇒* αXaβ
- Let nullable(X) be true when X can derive empty string ε

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

Figure 5: Top-down parsing table. You do not want more than one possibility in a cell.

- Abstract Syntax Tree (AST):
- Context-Free Grammars (CFG):
    - *Terminals → 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 4.
    - 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: $S \rightarrow S + x \mid S - x \mid x$. Here you don't know what to do when you see an $x \ldots$
* See figure 5 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, α is confirmed to be parsed, β is to be confirmed, $N \rightarrow \alpha.\beta$. Notice that it looks like a production rule, but with a dot somewhere in it.

- Item is reducible if β is empty. The right side of the dot is empty.

- ϵ-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 $X \rightarrow A\ B\ C$; 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 6 for parsing table, DFA for shift reduce grammar.
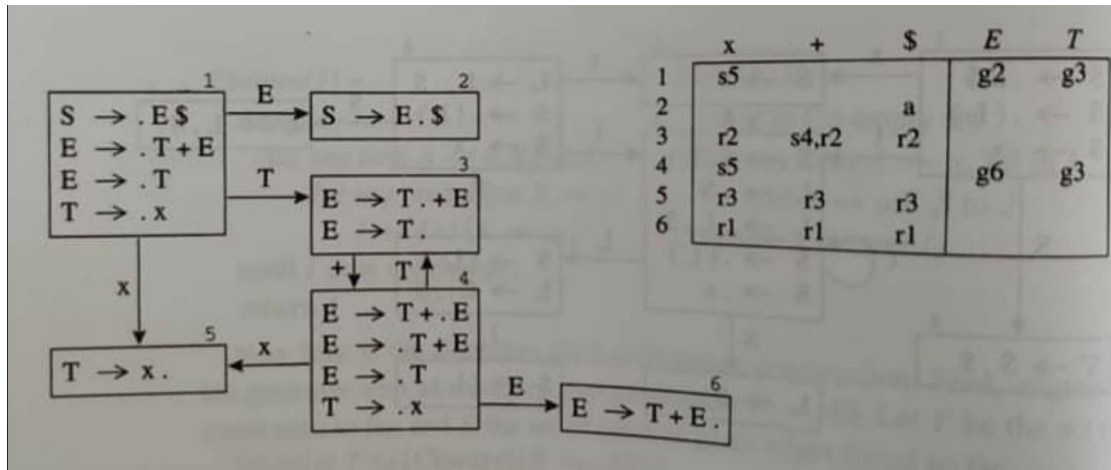
State 1:
```
S → .E $
E → .T+E
E → .T
T → .x
```
State 2:
```
S → E.$
```
State 3:
```
E → T.+E
E → T.
```
State 4:
```
E → T+.E
E → .T+E
E → .T
T → .x
```
State 5:
```
T → x.
```
State 6:
```
E → T+E.
```

	x	+	$	E	T
1	s5			g2	g3
2			a		
3	r2	s4,r2	r2		
4	s5			g6	g3
5	r3	r3	r3		
6	r1	r1	r1		

Figure 6: LR(0) shift/reduce conflict, parsing table and state DFA, sn: 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 $A \to x.By\ \{t\}$, where x and y are arbitrary strings of terminals and nonterminals and B is a nonterminal, you add an item of the form $B \to .w\ \{s\}$ for every production $B \to w$ 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.

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).

3.2.1   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