PLPDI Compilers: Revision Notes

Sam Barrett

January 9, 2021

Compiler books often focus on the task of compiling the C programming language. This was intentionally not the focus of this submodule.

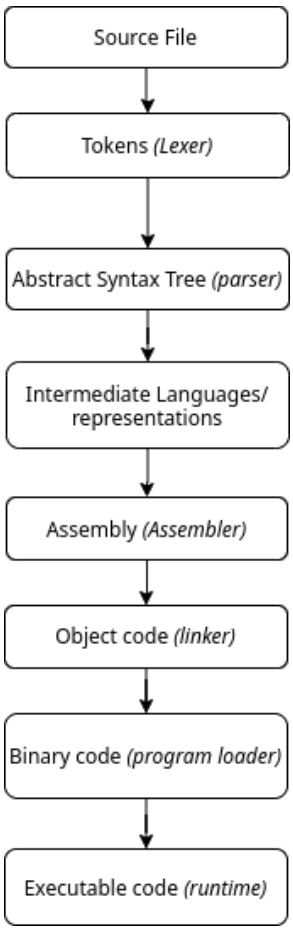


Figure 1: Compiler Flow

1 Lexical Analysis

Lexical analysis, the job performed by the *Lexer* is the first stage of compilation. It converts a source file written in a specific programming language into a string of *tokens*. Or alternatively, string $\implies$ list of "lexemes"

For example, the code:

```
fold (+) [1;2] 0 ⟹
                identifier open-round-bracket operator
                close-round-bracket open-square-bracket constant
                semicolon constant close-square-bracket constant
```

Lexemes are specified using regular expression which, in turn, are implemented using finite state automata.

For example, a simple Regular Expression capturing the *lexeme* of numbers:

$$num = \texttt{[+-]?[1-9][0-9]}^*$$

Backtracking is the *dumb* approach to parsing regular expressions. It is also known as the brute force method as in the worst case scenario, all possibilities could be tried before the approach *gives up*.

Such an expression which is tricky to parse using backtracking is:

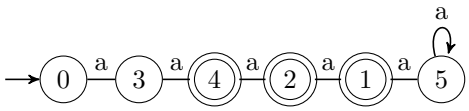
$$tricky = \texttt{a?a?aa}$$

Every occurrence of the *optional* (?) doubles the number of possible routes through a parse tree of this expression. Meaning an input such as `aaaaa` would cause an evaluator to explore every possible branch of this tree before determining that it is not captured.

1.1 Finite State Automata

A better approach is to use Finite state automata. This approach was originally proposed by Ken Thompson.

Using FSA our *tricky* regex can be converted to the following Deterministic Finite State Automaton or DFA.



However, the process to create a DFA is computationally expensive. Therefore, Thompson's algorithm instead operates over Non-deterministic finite state automata. It works by keeping track of all the positions in the DFA where we could be given the string we have seen so far.

2 Parsing

In this stage of the process where the sequence of tokens generated by the lexer is transformed into the Abstract Syntax Tree (ASG) of the program. An ASG is a record of the grammatical structure of the program and is essential to the correct interpretation of a program.

2.1 Grammars

A grammar is a set of rules. They are defined using Terminals and Non-Terminals. They are differentiated by the fact that during every application of a rule, the non-terminal occurring on the left-hand side (see below) is replaced by a sequence of terminals and non-terminals appearing on the right-hand side.

An example grammar could be:

$$E \to N | E + E | E * E$$

When used as a *production system* our rule(s) is(are) repeatedly applied to a designated starting symbol until we produce an expression comprised exclusively of terminal symbols.

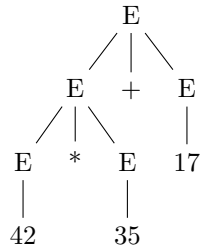
For example:

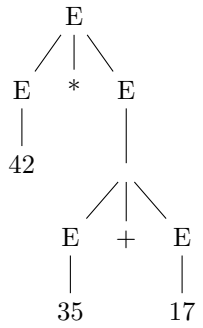
$$\begin{aligned} E &\to \underline{E} + E && \texttt{apply } E \to E + E \\ &\to \underline{E} * E + E && \texttt{apply } E \to E * E \\ &\to 42 * \underline{E} + E && \texttt{apply } E \to N \\ &\to 42 * 35 + \underline{E} && \texttt{apply } E \to N \\ &\to 42 * 35 + 17 && \texttt{apply } E \to N \end{aligned}$$

We can also run this process in reverse to convert a string of terminal symbols back to the starting non-terminal E.

We can however, recognise an issue with this process whereby, the set of possible operations at any given point is greater than 1. i.e. the process is <u>not</u> deterministic.

For example:





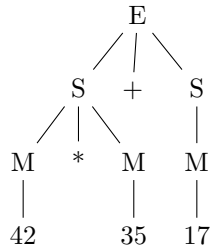
These trees are both valid under this grammar but they are not both valid mathematically. This is because the grammar does not embed the *strength* of the operators.

$$(42 * 35) + 17 \neq 42 * (35 + 17)$$

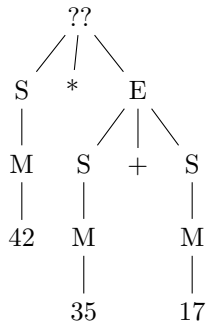
Ambiguity and non-determinism are some of the biggest issues faced when constructing a grammar indented for use as a production system or parsing. It has been shown that the ambiguity of context-free grammars is an un-decidable problem, i.e. there is no way to construct a program that can *decide* whether a given grammar is ambiguous. This leads to the designing of grammars more of an *art* relying on the experience of the creator and the use of heuristics.

We can re-write our original grammar in a non-ambiguous way to prevent the formulation of strings that can have conflicting parse trees.

$$\begin{aligned} E &\rightarrow S + S | S \\ S &\rightarrow M * M | M \\ M &\rightarrow (E) | N \end{aligned}$$



Above you can see that our valid example can be constructed using this new grammar but below you can see that the semantically invalid tree is now also structurally invalid:



2.2 Top-down parsing

2.2.1 Recursive Descent

Recursive descent means we start at the top with our starting non-terminal, in this case E. We then apply rules until we reach a string comprised entirely of terminals, if the terminals match the string we are trying to generate then the string is valid otherwise we backtrack and apply different rules. We repeat this process until we either exhaust all possible valid combinations of rules or we match the terminals.

From the section on Lexical Analysis we know that backtracking is rarely a good solution as it has very poor worst-case complexity (exponential).

2.2.2 LL(k) Parsing

There is an improved version of top-down parsing called LL(k) or left-to-right leftmost derivation with k tokens lookahead. The benefit of a *lookahead* is that it is able to disambiguate the application of the rule. The caveat is that the grammar has to be written in a compatible way for this system to work.

The LL(1) form of our grammar is formulated as such:

$$
\begin{aligned}
E &\to TE' \\
E' &\to +E|\varepsilon \\
T &\to FT' \\
T' &\to *T|\varepsilon \\
F &\to N|(E)
\end{aligned}
$$

Grammars of this type can be constructed algorithmically from an already deterministic grammar. **However,** not all grammars can be put into LL(1) form. It has been shown that it is undecidable whether given a grammar there is a fixed value of k s.t. the grammar can be put into LL(k) form.

2.2.3 Monadic Parsing

In order to avoid backtracking we build on the fact that we have seen that we can often replace time complexity with space complexity.

Simply speaking, in this approach we remember, inside of a data structure, the choices we have not made. i.e. in a list we store all the possible rules and evaluate all of them at the same time, discarding invalid trees until we either run out of trees or find one that is valid.

2.3 Bottom-up parsing:

2.3.1 LR(k): left-to-right rightmost derivation with k tokens lookahead (LALR)

Here we start at the *bottom* with our terminal string and apply rules until we reach our starting non-terminal. We again utilise k lookahead to disambiguate rules as we apply them, this allows us to discount a rule earlier if we see that it doesn't reduce to a required intermediary form.

There are a few problems which can arise when using LALR parsers.

One of these issues is known as a shift-reduce conflict. A typical example of a shift-reduce conflict is if-then-else in languages that permit both if-then and if-then-else syntactic structures. For example:

```
if x then if y then a else b
```

Is this mean to be processed as:

```
if x then{if y then a} else b
```

or

```
if x then{if y then a else b}
```

These are usually disambiguated by the parser mechanism preferring the *shift* to the *reduce*. Essentially, always trying to construct the longest possible parses.

A more problematic type of conflict is the reduce-reduce conflict.

$$\begin{aligned} \texttt{Seq} &\rightarrow \varepsilon \\ &\mid \texttt{Maybe} \\ &\mid \texttt{Seq}\ a \end{aligned}$$

$$\begin{aligned} \texttt{Maybe} &\rightarrow \varepsilon \\ &\mid a \end{aligned}$$

Here we can derive a with 2 different parse trees, either via the Seq structure or Maybe structure.

When you have one of these conflicts it is unclear which operation is going to happen and generally the execution order boils down to the order of the rules in the grammar definition.

Note: both of these conflicts are typically unavoidable in a large grammar, mitigating them is an art rather than a science.

Larger parsers are generally not written by hand and are instead generated using a **Parser generator** such as *YACC, Bison, Parsec, etc.*

3 Intermediate Representation

3.1 Abstract Syntax Trees (ASTs)

After the parse tree is created, it needs to be processed further so that it is easier to execute/compile. It is easier to initially consider interpretation. We want to, given an abstract syntax tree, interpret the expression that the Abstract syntax graph represents.

For example given the input string:

$$(1 + 2) + (3 + 4)$$

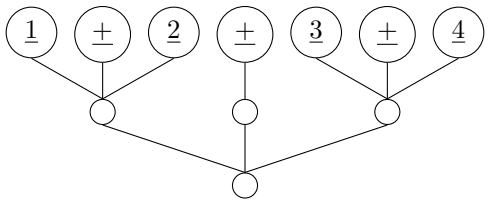
We token-ise it to get:

$$(\ \underline{1}\ \underline{+}\ \underline{2}\ )\ \underline{+}\ (\ \underline{3}\ \underline{+}\ \underline{4}\ )$$

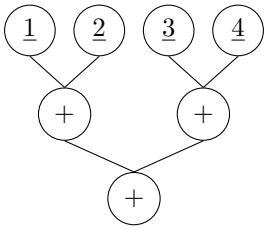
where underlined values are individual tokens.

We then construct a parse tree of this tokenised string:

Note: the names of the terminals/non-terminals is unimportant



Notice how we have removed the brackets; once the tree has been created, the brackets become implicit. We can also push the operators onto the parent nodes to produce an equivalent but prettier Abstract syntax tree:



We can show program execution as transformations on these Abstract Syntax Trees. How do we know what order to apply these transformations? We use a fixed traversal method such as Depth first left-right traversal.

The application of this form of traversal on our exemplar can be seen in Figure 2

ASTs do not cope well with variable names however, this is a big issue as variables are key in useful programming languages.

We will be focusing on **immutable** variables in the examples below.

```
let x = 1+2 in x + x
```

This is an example of a ternary statement, i.e. it has 3 components. It can be drawn as the following AST:

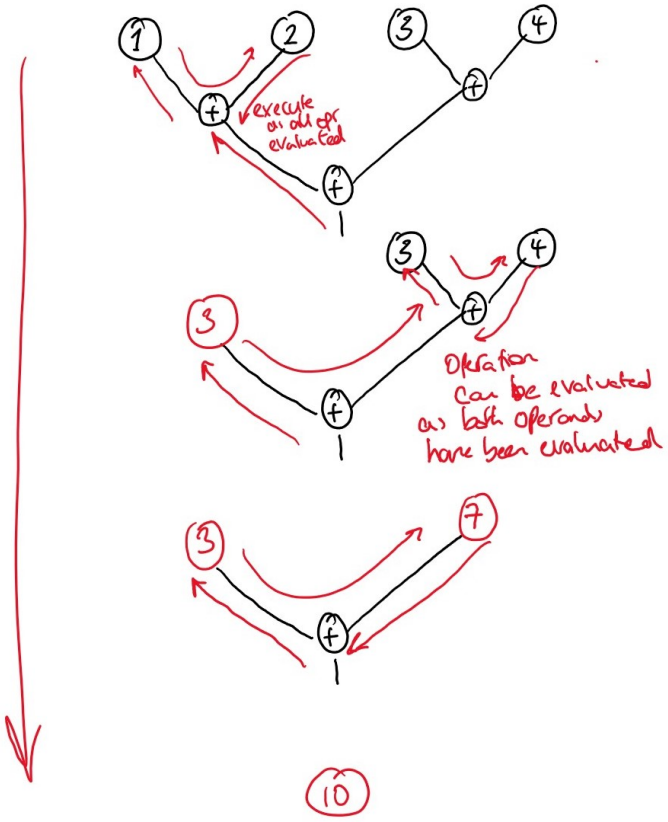
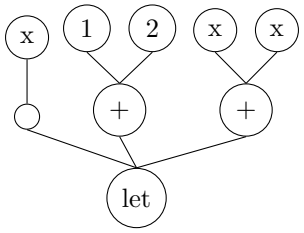
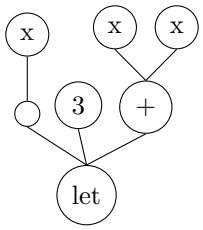


Figure 2: Depth-first L-R traversal



With this representation we are left with variable names on the leaves of the tree, with no reference back to their definition. If we were to evaluate this in the same way as we did previously we produce the following tree:



When we reach the first x reference, we are forced to backtrack through the tree in order to find it's associated value. This is extremely inefficient, especially in the case where we have nested variable usage, for example in:

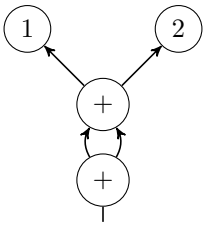
```
let x = 4+5 in let x = 3+2 in x+x
```

When evaluating this expression we must be careful about the scope of the variable(s) x as it has multiple definitions within the same expression.

We must find a better solution to this problem. We do this by converting our tree into a Directed acyclic graph (DAG), turning our AST into an Abstract Syntax Graph (ASG)

3.2 Abstract Syntax Graphs (ASGs)

We can draw the ASG of our original `let` expression as such:



We can now apply our Depth-first left-to-right evaluation order easily and efficiently. However, rememebr we are no longer performing a tree traversal but a **graph traversal**. The evaluation of this ASG can be seen below:

Converting from ASTs to ASGs introduces some compile-time overhead but allows for much more efficient execution of ones code.

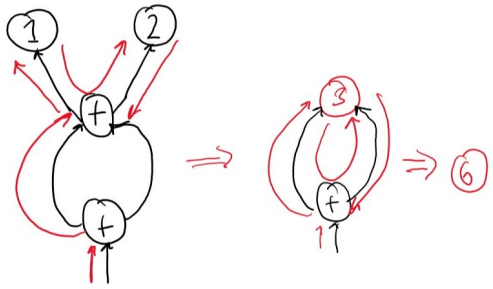


Figure 3: ASG Traversal

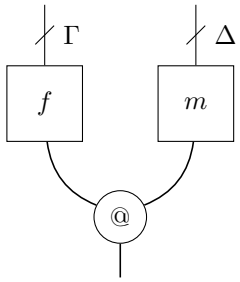
3.3 Hierarchical Abstract Syntax Graphs

Functions in the context of the λ-Calculus only have 2 operations we would need to be concerned with with respect to an abstract intermediate representation: 1) Abstraction 2) Application. How can we encode these into ASGs?

Function Application

$$\underbrace{x_0, \ldots, x_n}_{\Gamma}, \underbrace{y_0, \ldots, y_n}_{\Delta} \vdash f(m)$$

Here we represent the identifiers of f by Γ and the identifiers of m by Δ. We can consider function application as an operator. Therefore to construct an ASG of this expression, we construct the ASG of f and the ASG of m and combine them using an application operator, represented as @.



Here the stricken through lines represent many lines $(1 \ldots n)$

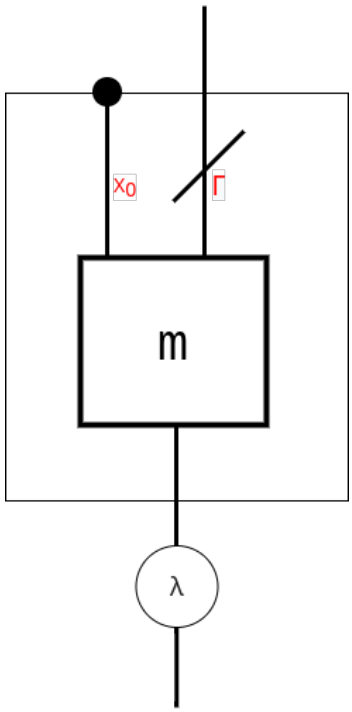
When we apply a function we carry out what is known as the *small* β rule. It can be formulated as:

$$(\lambda x.f)m \rightarrow f[m \backslash x]$$

This is very similar to the rules we have seen in operational semantics.

Function Abstraction

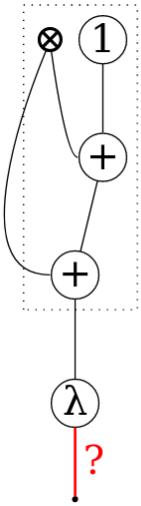
$$x_0, \underbrace{x_1, \ldots, x_n}_{\Gamma} \vdash \lambda x_0.m$$



In the above ASG you can see the representation of a λ-abstraction. The bound variable x_0 does not leave the scope of the abstraction (denoted by the outermost box).

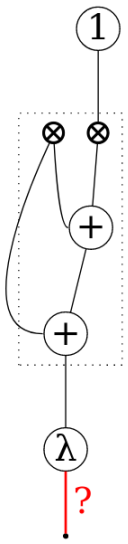
3.3.1 Examples

1. $$\lambda x.x + x + 1$$



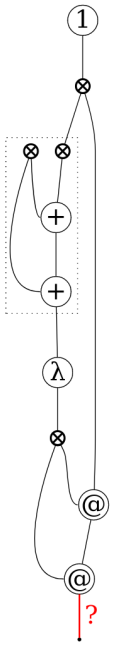
2.

$$\texttt{let}\ \ y = 1\ \ \texttt{in}\ \ \lambda x.x + x + y$$



3.

$$\texttt{let}\ \ y = 1\ \ \texttt{in let}\ \ f = \lambda x.x + x + y\ \ \texttt{in}\ \ f(f(y))$$



3.4 Note from Revision Lecture

We must be aware of the distinction between a compiler evaluation and optimisation.

Abstractly, a evaluation is a set of abstract syntax graph transformations applied according to a schedule. For a call-by-value programming language this takes the form of a depth first search traversal of the ASG with reductions applied on the path *back* up the tree towards the root.

A compiler optimisation is applying some transformations that make the code simpler and faster in an arbitrary order

A compiler optimisation is applying some transformations that make the code simpler and faster in an arbitrary order. Optimisations often have conditions that have to be met for them to be applied. An example of a compiler optimisation is closure conversion. Closures being anonymous functions.

In a closure conversion we want to *pull* inner (anonymous) functions into global scope. However, this raises an issue: what do we do with the variables of the closure which are bound in the enclosing function? We solve this using a notion of environment and transforming all functions in a uniform way.

4 Types

Types first appeared as an attempt to give a foundation of mathematics using the λ-Calculus via the Curry-Howard correspondence whereby a proposition corresponds to a type and a program corresponds to a proof.

Initially, the point of types in the Simply typed λ-Calculus was to ensure that a given program will always terminate. Modern usages are to prevent runtime errors in programs.

In the Curry-Howard correspondence, we can consider a value $u : T$ as a predicate $T(u)$ which states that *'u has type T'*. This is what is known as a *type judgement*, type judgements are proved using logically formulated typing rules.

4.1 Type Checking

Given a program, u which takes typed variables $x_0, \ldots, x_n$ we can perform a type judgement that u has type T

$$x_0 : T_0, \ldots, x_n : T_n \vdash u : T \quad (\equiv) \quad \Gamma \vdash u : T$$

$$\frac{\Gamma \vdash u : T \to T' \quad \Gamma \vdash v : T}{\Gamma \vdash u(v) : T'} \text{ Implication elimination}$$

$$\frac{\Gamma, x : T \vdash u : T'}{\Gamma \vdash \lambda x.u : T \to T'} \text{ Implication introduction}$$

Here it is the parallels between logical implication and functions as expressed by the Curry-Howard correspondence are clear.

We can extend this to work for other types such as Product and Sum types. For product types we have the rules:

$$\frac{\Gamma \vdash u : T_1 \times T_2}{\Gamma \vdash proj_i(u) : T_i} \text{ Conjunction Elimination}$$

$$\frac{\Gamma \vdash u_i : T_i}{\Gamma \vdash (u_1, u_2) : T_1 \times T_2} \text{ Conjunction Introduction}$$

Using this pattern we can define the function types for references:

$$\begin{aligned} ref_T &: T \to \texttt{ref}(T) \\ assg_T &: \texttt{ref}(T) \to T \to () \\ deref_T &: \texttt{ref}(T) \to T \end{aligned}$$

We can go on to define increasingly complex types with this pattern, however, not all languages implement these. One such example is **dependant types** which are available in proof assistants such as Agda and Idris. They allow for the embedding of conditions into the type of a function, for instance the function "nth" can be defined simply as:

```
n : Int -> (xs : 'a List) -> 'a
```

However, this allows for runtime errors, for instance in the case where `n > length xs == true` the function would crash.

In languages with dependant types we can embed the proof that this isn't true in the type of the function:

```
n : Int -> (xs : 'a List) -> (n < length xs) -> 'a
```

4.2 Type Inference

Type inference is the process whereby given a term, we have to decide whether it is possible to assign types to variables so that according to our typing rules our term will now typecheck.

There are various methods such as the Hindley-Milner algorithm which works by collecting a set of *equations* from the AST created by denoted types with variables and solving the system of equations using the principals of reduction and unification.

The Hindley-Milner algorithm in the worst-case scenario has exponential time complexity. However it's average case is more acceptable.

For examples of constructing the ASTs mentioned above, see the video lecture.

5 Assignment

Programming languages that have operations that are not representable in the λ-Calculus have what are known as *side effects*. Languages that go without these features are called *pure*.

However, if you want to add machine dependant operations such as assignment or IO the language becomes *effectful* or *impure*

Assignment

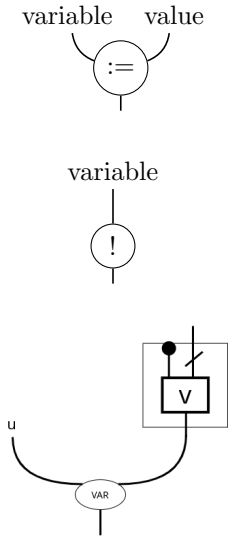
```
x := u
x = x + 1 == x = !x + 1 //true
```

Above you can see an example of assignment within an imperative language. The second line shows how most language add implicit de-reference when working with variable values.

More abstractly, this can be written:

```
let x = U in V
```

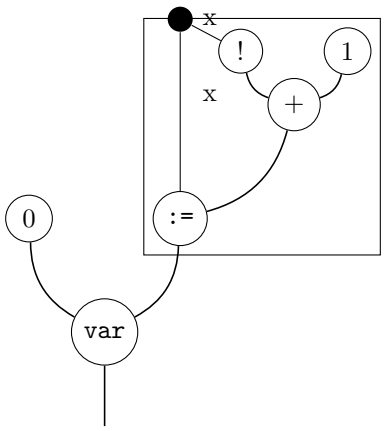
Where U is our assigned value and V is the block in which it is in scope. This can be represented as the following AST:



Above you can see that we represent a variable, v as a *thunk*. This is an alternative use for thunks, we previously used them when constructing functions, notice in this case we omit the bottom λ node.

Example

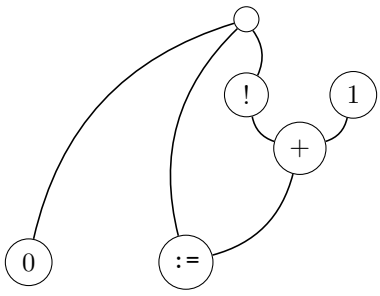
```
var x = 0 in x := !x + 1
```



Parsing this depth-first left-to-right we find values on both sides of our `var` operation. Informally, our rewrite operation will be:

- allocate/ create variable
- initialise to 0
- bind to x

This produces the following ASG:



6 Code Generation

'Abstract machines give you the cake.'
'Compilers give you the recipe.'
—Dan Ghica

Only touched on briefly due to complexity. Essentially, code generation is the final process of compilation. If Abstract machines execute the program, Compilers (via code generation) say what to do.

7 Tips for the Exam

1. Very similar to the summative assignment.
2. Use *SPARTAN*