LINGI2132

Languages and Translators

Emilyen Laffineur

Teacher :
Nicolas Laurent

Assistant:
Mathieu Jadin & Alexandre Dubray

Ecole Polytechnique de Louvain
Universtié catholique de Louvain
Belgium
Scholar year : 2020-2021

Contents

Chapter 1

Introduction

This class is about compiler. The goal of compiler is to implement a programming language. It take source code and make it executable (or pass it to a program that execute it). We may think that language like Python or JavaScript does not use compilers however they use it as well! Even if they are dynamical. Note that this class is not about programming paradigms, it's truly about implementing languages in general.

We will learn about compilers and their parts, allowing us to understand how languages works. We will also learn some useful patterns (Domain Specific Language (DSL), trees, interpreter pattern).

Chapter 2

Compilation Pipeline

Here is the classical compiler pipeline :

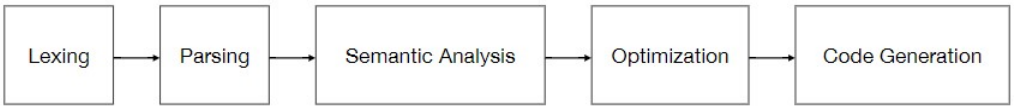

Figure 2.1: Compiler pipeline

We will go through each step.

2.1 Lexing

Definition 2.1.1 (Lexing)**.** It's a lexical analysis, tokenization, scanning. It take the text and produce tokens (= "words"). e.g : foo = bar + 42 will produce |foo| = |bar| + |42|.

2.2 Parsing

Definition 2.2.1 (Parsing)**.** It take the output of lexing (tokens) and extract it's structure (by producing an Abstract Syntax Tree (AST)). It also catches syntax errors (like missing parenthesis around if in Java). Other kind of errors (like something that is not defined by the language) is not a syntax error, thus, the parser cannot catch it.

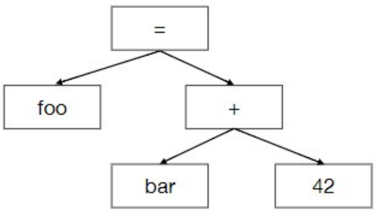

Figure 2.2: Abstract Syntax Tree

Note that lexing is optional while parsing (called scanner-less parser) for others parsing is actually lexing AND parsing.

2.3 Semantic Analysis

It consists of a series of check that is perform on the AST, two mains are Name binding et type checking.

2.3.1 Type checking

Definition 2.3.1 (Type checking)**.** Verify that the type passed is what is expected.

```
void print(String str)
print(42)
```

Definition 2.3.2 (Type inference)**.** Let the compiler guess the type of the variable given the context.

Note that type checking is inference. When :

```
String x = "foobar"
```

is used, it first need to guess what is "foobar", then check the consistency with the type.

Sometimes, we need to know the return value's type of a function to perform the check. That's what Name Binding is.

2.3.2 Name Binding

Definition 2.3.3 (Name binding)**.** Allow us to infer the type of a variable given the return type of a function.

Note that it becomes much more complex with OOP.

2.3.3 Rest

It also performs flow check (missing return for example), access control (public, private, etc.).

2.4 Optimization

Actually, the pipeline we saw in figure 2.1 it's only the old school C compiler. New C compiler, do optimization on the produced machine code (LLVM, Low Level Virtual Machine)! Do not get confuse, it does not use a virtual machine at all... LLVM generate IR (intermediate representation), optimize it and then generate machine code.

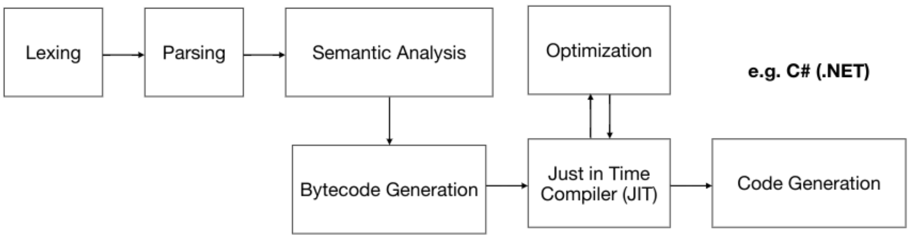

Figure 2.3: C# pipeline

This one generate bytecode (Java name), that use usable but that won't be directly use by the machine. After the generation of the bytecode, it is passed to a Just In Time Compiler (JIT).

What is done until JIT is done at compilation time, JIT and the following is done while running the program!

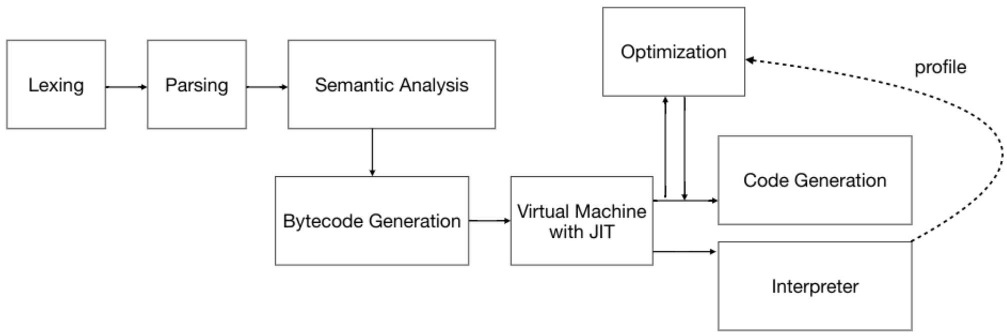

Figure 2.4: Java and Python pipeline

That's the architecture that most recent languages uses. The bytecode is passed to a VM executing a JIT. This VM can generate machine code (code generation) or pass it to in interpreter which will run a runtime profile that is much better for optimization! That the reason why Java can be as fast (or faster) the C. Pypy (Python), Java, TruffleRuby use this kind of architecture!

We could also not optimize it! We could directly use the Tree-Walk interpreter, Ruby before 1.9 did that. CPython (standard one), and Ruby generate the bytecode and run it in a VM.

2.4.1 Tree-Walk Interpreter

Let's take the following program :

```
var foo = 42 + 52
print(foo)
```

The AST is the following :

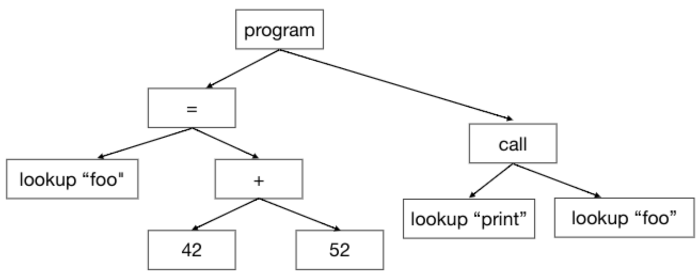

Figure 2.5: Resulting AST

(Note that print has is own AST). We have the scope, containing the value of foo saved in the store and the definition of print. The procedure in order to execute is recursive, it will lookup for foo, perform the addition, store it. Lookup at the print definition, lookup for foo and foo will be passed as a parameter to print.

2.5 Code Generation

It could be machine or bytecode generation. Example of Machine Code (e.g. x64) :

```
int square(int num) {
    return num*num
}
```

Becomes :

```
square(int):
    push rbp
    mov rbp, rsp
    mov DWORD PTR [rpb−4], edi
    mov eax, DWORD PTR [rpb−4]
    imul eax, eax
    pop rbp
    ret
With optimization it becomes :
    mov eax, edi
    imul eax, edi
    ret
```

While bytecode generation is:

```
public class Hello {
    int square(int num) {
        return num*num
    }
}
```

Which becomes (stacked base instead of registry based as before):

```
public class Hello {
    public <init>()V
       L0 LINENUMBER 1 L0
        ALOAD 0
        INVOKESPECIAL java/lang/Object.<init> ()V
        RETURN
       L1
        LOCALVARIABLE this LHello; L0 L1 0
        MAXSTACK = 1
        MAXLOCALS = 1
       square(I)I
        L0
          LINENUMBER 3 L0
          ILOAD 1
          ILOAD 1
          IMUL
          IRETURN
        L1
          LOCALVARIABLE this LHello; L0 L1 0
          LOCALVARIABLE num I L0 L1 1
          MAXSTACK = 2
          MAXLOCALS = 2}
```

2.6 Optimizations

As we have seen, we can optimize on tree or on target code. They are two types of optimization that are the base of everything :

- Inlining : pulling a function in another one
- Partial evaluation : propagating known information (e.g constant-folding)

Some other exists like loop unrolling, etc.

2.6.1 Inlining

Take the following program :

```
var foo = false
int a(){
    if (foo) return somethingLongAndBoring()
    else return b(true)+1
}
int b(boolean bar) {
    return bar ? 42 : 52
}
```

If we optimize the function a. It we online somethingLongAndBoring and b. However, if we do this and do not enter the condition, this could be bad for cache locality. Indeed, it will lead to a cache miss, with a lot of code, the time to retrieve the actual code will be huge. However, we know that foo is always false so we can get rid of the first condition! Then we can inline b. Thanks to the inlining of b, then constant folding it will lead on only this :

```
int a(){
    return 43;
}
```

Which is much more efficient! Thanks to that we can see that there is a complementary between constant folding and inlining.

Chapter 3

Formal Grammars

3.1 Language vs Grammar

Definition 3.1.1 (Grammar)**.** The (formal) definition (*description*) of a language

Definition 3.1.2 (Language)**.** A (potentially infinite) *set* of sentences, in programming language, a sentence is a source file, REPL expression, etc.

Definition 3.1.3 (Alphabet)**.** Composed of tokens, lexemes.

Let's take an example which is JSON. The language is all valid JSON expression. Sentences : e.g.

```
{
    "version": 17,
    "bundles": [
        { "name" : "org.graalvm.component.installer.Bundle" },
        { "name" : "org.graalvm.component.installer.commands.Bundle" },
        { "name" : "org.graalvm.component.installer.remote.Bundle" },
        { "name" : "org.graalvm.component.installer.os.Bundle" }
    ]
}
```

Figure 3.1: JSON Sentence

Grammar :

```
VALUE    ::= STRINGLIT / NUMBER / OBJECT / ARRAY
OBJECT   ::= "{" (PAIR ("," PAIR)* )? "}"
PAIR     ::= STRINGLIT ":" VALUE
ARRAY    ::= "{" (VALUE ("," VALUE)* )? "}"
```

Alphabet (Tokens)

Figure 3.2: JSON Grammar

3.2 Grammar notation

3.2.1 PEG

Parsing Expression Grammar. The "?" denotes optional, "*" means 0 or more times.

```
VALUE   ::= STRINGLIT / NUMBER / OBJECT / ARRAY
OBJECT  ::= "{" (PAIR ("," PAIR)* )? "}"
PAIR    ::= STRINGLIT ":" VALUE
ARRAY   ::= "{" (VALUE ("," VALUE)* )? "}"
```

Figure 3.3: PEG notation

PEG is a grammar formalism and a notation.

Definition 3.2.1 (Formalism)**.** Mathematical system to define the language (set of sentences)

Definition 3.2.2 (Notation)**.** A way to denote a grammar to be interpreted by the formalism.

3.2.2 EBNF and BNF

Note that (E)BNF is a notation for CFG (Context-Free Grammars).

EBNF

Extended Backus-Naur Form. "[]" means optional, "{}" means 1 or more times, so in order to make "0 or more" with have to combine symbols.

```
VALUE   ::= STRINGLIT | NUMBER | OBJECT | ARRAY
OBJECT  ::= "{" [ PAIR [{ "," PAIR }] ] "}"
PAIR    ::= STRINGLIT ":" VALUE
ARRAY   ::= "{" (VALUE [{ " ," VALUE }] ] "}"
```

Figure 3.4: EBNF notation

BNF

Backus-Naur Form. It is at the base of EBNF, it does not contain "{}" or "[]", so it uses recursion and a lot more rules. Note that ϵ is the mathematical notation for "nothing"

```
VALUE   ::= STRINGLIT | NUMBER | OBJECT | ARRAY
OBJECT  ::= "{" PAIRS? "}"
PAIRS?  ::= ε | PAIRS
PAIRS   ::= PAIR TPAIRS
TPAIRS  ::= ε | "," PAIRS
PAIR    ::= STRINGLIT ":" VALUE
ARRAY   ::= "[" VALUES? "]"
VALUES? ::= ε | VALUES
VALUES  ::= VALUE TVALUES
TVALUES ::= ε | "," VALUES
```

Figure 3.5: BNF notation

3.2.3 Summary

CFG and PEG are different formalism. In some case, PEG and CFG can be the same (like in JSON) because it is a very simple definition. By it is not true in general. Sometimes, we can use PEG notations like +, *, ? in CFGs (coming from REGEx, which are related to CFGs.). Actually we can use any notation that we want, we just need to define it!

Chapter 4

Parsers

Definition 4.0.1 (Parser)**.** A parser is a program that :

- Accepts/rejects input a sentence in the language (recognizer)
- Extracts a(n) (abstract) syntax tree

4.1 Parsing Tools

A parsing tool lets you create parsers. We can denote two kinds : Generator an Library. Note that is just a generalization. The main distinction to retain is the generation vs interpretation.

Parsing tools are often called "parsers", it is acceptable, but incorrect, parsing tools create and/or run parsers.

Parsing tools are compilers!

- Language = grammar notation
- Code generation (parser generators) or interpretation (parsing libraries) (depending on the used tool)
- Domain Specific Language (DSL) (depending on how we define it. DSL is opposed to Turing-Complete language like Java, Ruby, Python, etc. A DSL could be XML, JSON, Grammar, SQL, etc.)

4.1.1 Parsing Generator

Note : in the following figure, all solid boxes denote programs.

Definition 4.1.1 (Parsing Generator)**.** A Parsing Generator is :

- Outputs a parser program (typically source code)
- Typically a command line tool
- Typically uses own grammar language

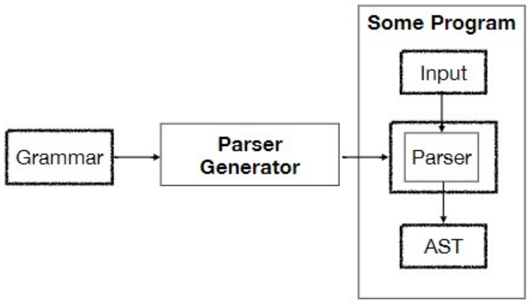

Figure 4.1: Parser Generator

4.1.2 Parsing Library

Definition 4.1.2 (Parsing Library)**.** A Parsing Library is :

- Let us "interpret" a grammar
- Grammar typically defined with a DSL.

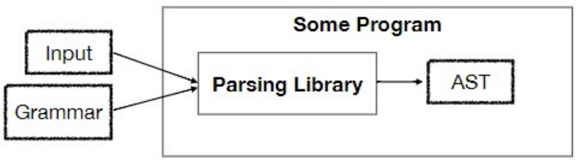

Figure 4.2: Parser Library

Note that input AND grammar could be in the "Some Program" box.

4.2 (Abstract) Syntax Tree

4.2.1 Syntax Tree

Remember the notation of figure 3.2 and 3.1. To make it easier for Syntax Trees we will use the figure 3.5. Doing so, we can extract the following ST :

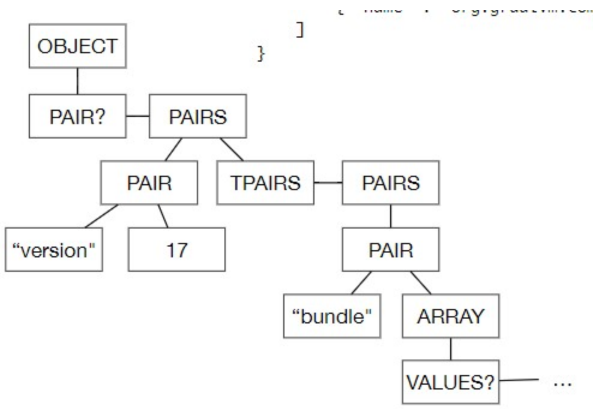

Figure 4.3: Syntax Tree

Syntax Tree is the following : for each phrase in the sentence we follow the grammar. It is the only thing we can do!

4.2.2 Abstract Syntax Tree

Here is the same Syntax Tree but abstracted :

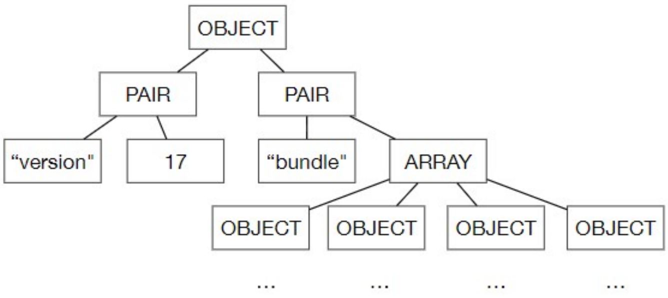

Figure 4.4: Abstract Syntax Tree

As we can see, it is much more clear than the classical Syntax Tree.

4.3 Summary 1

Parsing Tools that allow us to use grammar definition like PEG or EBNF (with more symbols that denotes more explicit stuffs) will generally produce better Syntax Tree than the others. However AST let us decide precisely what we want as node/leaves in the tree. Let's take an example using Java :

```
list.forEach(x −> System.out.println(x));
list.reduce((x,y) −> x + y);
```

As we can see, the difference is that with 1 identifier we do not need parenthesis. That must be take into account when generating the Syntax Tree. It would lead to the following Syntax Trees :

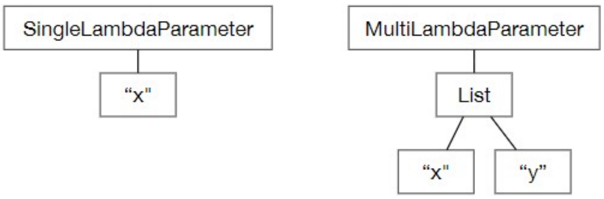

Figure 4.5: First ST

However, as we have seen, parsing is just the start of the pipeline! We have to do a lot of work with it (check for errors, generate bytecode, etc.). With this kind of Syntax Tree, we would have to deal with different kind of node for the same lambda principle. Using AST we can have the following :

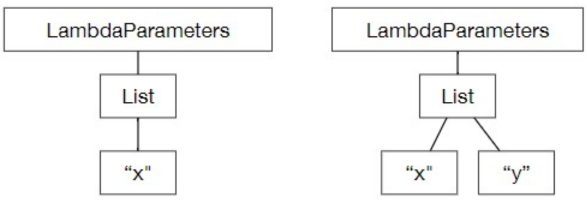

Figure 4.6: Second AST

Which allow us to use the same code to deal with both!

4.4 Coding a parser

General principles :

- One function per symbol (non-terminals & terminals);
- Model in the input as globally accessible character array;
- The current input position is globally accessible;
- Calling a function = attempting to parse the symbol it denotes at the current input position;
- A parsing function returns :
  - true if the symbol was matched, updating the input position past the matched input.
  - false if they failed to match, the input remains unchanged.

See parser.java file. Note that, in order to be sure our implementation really work, we can use the debugger and check for the pos cursor at the end, or to go further we could also inspect the parse tree if we have build one.

As we can see, the parser we wrote is verbose and cumbersome. Also it does not have any bells or whistles, if it fails it just fails, it does not tell us where it fails or why. What we have done is implementing a PEG semantics for the grammar (hint, a* a is empty). Backtracking in PEG is to reset the counter before the start of the sequence. CFG also have a backtracking technique but it is not the same.

We can also add AST to our parser. In order to do that in Java, we just need to add a Dequeue and modify only a few methods. See parser_ast.java for that.

4.5 Parser Combinator

The parser we have implemented from PEG grammar. As we have seen, this parsers have a lot of duplication (each "expression" is handled the same way). The idea to solve that is to build an AST where each node is an "expression" and interpret it.

The idea, is that each combinator is a node/parsing expression in the AST. See file Combinators.java. Using combinators allow us to cut down on verbosity & code duplication (across all grammars and within a specific grammar). However it still have some drawbacks :

- Harder to debug
- It's slower because of megamorphism (we'll come back later on this)

Note that theses issues can be eliminated by using combinators for code generation. Good frameworks (like Autumn) will mitigate usability issues.

Chapter 5

PEG & CFG Semantics

5.1 Recap

Definition 5.1.1 (Grammar)**.** The (formal) definition of a language

Definition 5.1.2 (Language)**.** A (potentially infinite) set of sentences

Definition 5.1.3 (Parser)**.** Recognizes a sentence in the language + extract a syntax tree

Definition 5.1.4 (Parsing Tool)**.** Generate and/or runs parsers

Two types of formalism :

- CFG (Context-Free-Grammar)
- PEF (Parsing Expression Grammar)

Notations :

- (E)BNF (usually for CFG)
- PEG Notation (usually for PEG)

Definition 5.1.5 (Non-Terminal)**.** Things that we define

Definition 5.1.6 (Terminals)**.** Tokens, strings, etc. that cannot be extended further.

5.2 Context-Free Grammars

Usually a CFG is defined as a tuple of 4 components Grammar $= (N, \Sigma, P, S)$:

- N: Non-Terminals
- Σ: Alphabet (Terminals)
- P: Production Rules ($P : N \to (\Sigma \cup N)*$)
- Starting Symbol ($S \in N$)

In order to have a CFG from a "full" grammar, we first need to replace all syntactic sugars by the complete recursive rules, then eliminate all choices by introducing as many rules as we have choices.

5.2.1 Semantics Derivation

- The language defined by a CFG is the set of all sentences that can be derived from its rules
- Start from the start symbol, replace it by the right-hand side of one its production
- At each step, replace a non-terminal from the current string symbol by its definition (until no non-terminals are left in the string)
- Any terminal, the order does not matter

Note that by doing that, we define the language, not a parsing algorithm and we're doing it in a generative way grammar → sentences.

Also, if we would have used characters derivation instead of tokens we would have need to continue to derive which would of potentially lead to an infinite derivation. So, we can say that for a sentence to be part of a language in need to be derivable however we cannot describe the all derivation table because in many case, it would be infinite.

5.3 PEG Semantics

We can see in the slide (9 of PEG - CFG semantics's pdf) the use of a top-down recursive descent parsers that produce production rules (with lookahead operator as an exception).

5.4 PEG vs CFG

5.4.1 Semantics

CFG are generative (their semantics is given by constructing the language set by the grammar through derivation). On the other hand PEG are recognition based : a sentence is in the language defined by a PEG grammar only if it is recognized by the language recognizer. In order to formalize PEG grammar we need to formalize the recognizer for the grammar.

These two approaches are very different in the mathematical point of view. Also, CFG is easier to defined in the mathematical language, on the other hand the PEG is very much related to the practice.

5.4.2 PEG vs CFG

- CFG has unordered choices, while derivating we can pick any non-terminal we want
- PEG has ordered choices, the first matching is the "correct" one

5.4.3 The differences

Unordered and ordered has a consequence for parsing algorithm. PEG does not test anything after a fail (prefix capture) if the parse failed while CFG can make an other choice!

PEG: Single parse Rule

Once a choice have been made, we never visit it again. As repetition can be desugared to choice it has an impact on them too. Repetition are greedy. (A ::= a* a is empty as a* will consume everything).

Backtracking

Let's take the following grammar :

- A ::= B y z
- B ::= v | x y | x
- input: "xyz"

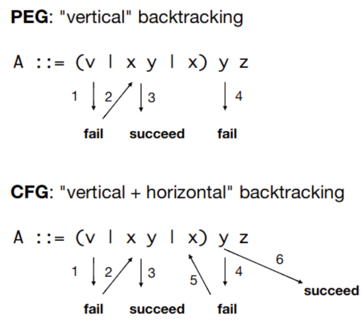

Figure 5.1: PEG vs CFG backtracking

This is not the real working way, be it does illustrate the principle. Indeed doing that would lead to exponential time complexity. CFG has horizontal backtracking thanks to is possibility to backtrack on x after trying y.

Ambiguity

It seems that CFG are always better because of what we have seen before. However there is a flip-side : ambiguity. By construction PEG cannot suffer from ambiguity.

As a remainder :

- Prefix capture (The rule B will consume all y)
  - A ::= B y z
  - B ::= x y | z
- Ambiguity (B can be xy or x, C can be z or yz):
  - A ::= B C
  - B ::= x y | x
  - C ::= z | yz

Ambiguity make the AST creation harder and also has impact on the parser performances.

Performances

The best parsing algorithm for CFG are $\mathcal{O}(n^3)$, deterministic parts of the grammar run in $\mathcal{O}(n)$ (most of useful grammar are deterministic).

For PEGs, the regular algorithm is exponential (in theory). Still, it is almost impossible to write an $\mathcal{O}(x^n)$ exponential parser. However, one often use operation is very inefficient : infix.

An example can be this one :

- S ::= P '+' S | P '-' S | P

- P ::= N '*' P | N '/' P | N
- N ::= [0-9]+

(Note that PEG does not allow left recursion by definition.) Let's assume we have a parseN function, if we try to parse "42", parse N will be called 9 times! 3 times for P, 3 times for S, 3x3=9. In general : $\mathcal{O}((P+1)^L)$ times with L : precedence levels (2 here), P : operators at each level (3 here).

A solution for that can be found, for example in Autumn we can write :

```
rule P = left_expression()
                .operand(N)
                .operand('*')
                .operand('/'); //Same for S however, operand is P in S, for precedence
```

This rewrite the grammar as :

- S ::= P ('+' S | '-' S)*
- P ::= N ('*' P | '/' P)*
- N ::= [0-9]+

P and N will be only called once! Performances are good thanks to that! However we can note that the parse tree won't be nice (not a problem with Autumn as we build an AST explicitly (we give a function to create the nodes)). If we were not using Autumn, we should create the parse tree like that and then, rewrite it

Packrat Parsers The single parser rule make memoization very easy! Packrat Parser are PEG parser with memoization. However, some practical experiments have been done in Java shown it is slower. Unless maybe if your language is very slow, or unless your infix expressions are improperly implemented.

Expressivness

- Some PEGs definition cannot be defined with CFG (A ::= $a^n b^n c^n$ for any same n we want any a b c)
- Some CFGs definition cannot be defined with PEGs (A ::= a A a | b A b | a | b | ϵ)
- Traditional PEG can't use left recursion (use repetition or Autumn to solve that, as they are the only two big case where we need left recursion)

PEG : Lookahead operators

- &<expression>, succeeds if the expression succeeds but does not consume any input
- !<expression>, succeeds if the expression fails, does not consume any input
- In theory, &X == !!X (not true in Autumn)

PEG misc

PEGs are often use in scannerless parsing (without lexer), ordered and lookahead are very useful at lexical level. PEG allow us to define "reserved works", hence lexing can still be advantageous.

PEGs are easy to extends thanks to recursive descent (with new combinators).

5.5 Summary

PEG :

- Intentional language = sentences recognized
- Similar to handwritten top down recursive descent parsers
- Desugared to CFG + lookahead
- Single parse rule
- Suffer from prefix capture
- Vertical backtracking
- Deterministic
- Ordered
- Potentially exponential, in practice largely linear (careful with infix)

CFG :

- Extensional language = set of sentences obtained by derivation
- Suffer from ambiguity
- Vertical and horizontal backtracking
- Non deterministic
- Unordered
- $\mathcal{O}(n^3)$ but often linear in practice

In the end, both formalisms are good enough, the real difference is tooling, what is available in the language we want to use, ease of use, features, performances, etc.

Chapter 6

Chomsky's Hierarchy

6.1 The hierarchy

Chomsky which is a famous linguist also known for his political engagement has defined a hierarchy of grammar :

- Type 3: Regular grammars (regular expressions, single non terminal at the end)
  - $P : N \rightarrow \Sigma * N$
- Type 2: Context-Free grammars (non terminal string of symbols)
  - $P : N \rightarrow (\Sigma \cup N)*$
- Type 1: Context-Sensitive grammars (alpha and beta represent the context)
  - $P : \alpha N \beta \rightarrow \alpha \gamma \beta$ where $(\alpha, \beta, \gamma \in (\Sigma \cup N)*)$
- Type 0: Unrestricted grammars
  - $P : (\Sigma \cup N)+ \rightarrow (\Sigma \cup N)*$

6.2 Regular vs CFGs

In regular we have a final non-terminal makes regular languages capable of expression repetition and optionality. Regular cannot express nesting.

6.3 Type 0 and 1

Let's be honnest, they are pretty much useless. Still, the principle behind context sensitivity is useful! Indeed, is func((T) * x) a multiplication or a cast? However CSGs are bad for that (not like Autumn).

6.4 Automaton mapping

Definition 6.4.1 (Automaton)**.** Formalisation of an abstract machine

A nice thing with the hierarchy is that it can be mapped to automaton :

- Type 3 (Regular grammars) : Deterministic Finite Automaton

- Type 2 (Context-Free grammar) : Nondeterministic (can have multiple transition) Pushdown (use a stack, a transition can push on it or look at it to take a decision) Automaton
- Type 1 (Context-Sensitive grammar) : Linear Bounded Automaton (Finite-tape Turing Machine)
- Type 0 (Unrestricted grammar) : Turing machine

Chapter 7

Lexing with regular expression

7.1 Importance of Lexing

7.1.1 Whitespace Handling

This chapter will show why lexing is important in addition to parsing. A simple example is whitespace (WS) handling. If we look at the grammar we have defined before (N, P, S) the expression " 1 + 1 " is not accepted because of whitespace. In order to have WS accepted we can use the following rule : $S ::= S' +' WS * P|S' -' WS * P|P$ with WS being space, \t or \n but it is not nice and can quickly become a mess. A nicer approach would be to define WS as before and to define new rules for operators like $PLUS ::=' +'WS*$.

7.1.2 CFG limitation

As we have seen before CFG does not allow reserved work like if.

7.1.3 Performances

If we don't use lexing, we need to work on each character, using a lexer we can build a token tree which can be 20 smaller. This is especially true if there are fixed overheads (combinators). It is typically developed as a loop with switch statement. It is simple enough that we can write it by hand or even better generate it (regular expressions)!

7.2 Lexers with RE

The idea is that we want to match a prefix of the (remainder of) input that matches the RE. There are two types of RE flavors :

- Standard (equivalent to regular grammars) : can be matched in $\mathcal{O}(n)$ (n=input size)
- Language specific (Perl-Compatible Regular Expression) : often in $\mathcal{O}(n)$ but some features can make it exponential (even without using them). E.g : backreferencing (capture something we have match and try to match it again).

Some experiment on this subject have been made which show the time needed by both techniques to match some rules. As we can see on the next figure the difference is huge. Note that, the y axes is not the same, with PCRE we're speaking in seconds while in the other approach we are on μs which is way smaller!

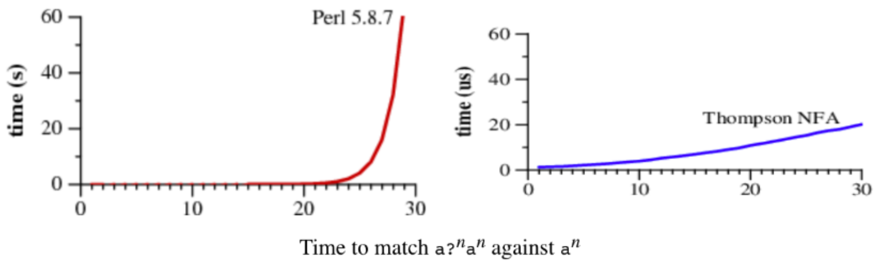

Figure 7.1: Regular vs PCRE

However we can ask ourself if this experiment is really realistic. As we have seen, it *can* be exponential in some case but are these case something that can really happen? This question is not so easy to answer if we make some research (Pr. did) we can maybe find one grammar example that is both useful and exponential.

Let's now image that matching a RE is done in $\mathcal{O}(n)$, what is the complexity of lexing? It is $\mathcal{O}(n^2)$! Why ? Because in the worst case we would need to scan to the end of the input (n) for each token. An example would be : $(a|a*x)$ with a^n as input. This is a contrived example which is not a problem in practice! This is also a nice example of theory vs practice.

7.3 Building the DFA

We have seen before that regular grammars can be recognized by a Deterministic Finite Automaton. First let's look at the DFA for rule $Tokens ::= [0-9]+|[a-z]|"if"$. The resulting automaton is this one :

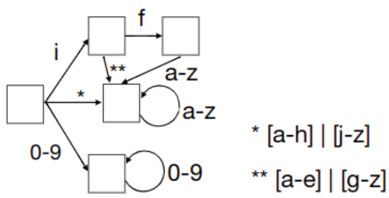

Figure 7.2: DFA example

In order to build the DFA, we first need to translate the regular expression to a NFA (Nondeterministic Finite Automaton). Then we have two options :

1. - Transform the NFA into a DFA
   - Minimize the DFA
2. - Simulate DFA with a NFA
   - Exactly the same thing, but lazily

7.4 NFA

First of all let's see an example of NFA, the left part is showing the NFA whereas the right part is the equivalent DFA. As we can see, the difference is that in NFA we can have multiple transition that use the same letter.

Figure 7.3: NFA example

If we try to match a sequence in a NFA, we have two ways :

- Keeping a single pointer that rollback at the beginning if the match failed
- Keeping multiple pointer that go in parallel.

In DFA the complexity is $\mathcal{O}(n)$ well, in fact it's more $\mathcal{O}(nm^2)$ (at least in theory) where m = number of states and represent the possibility where every state is connected to each other. In practice, NFA use repeated work and more complex data structures. That, plus the fact that DFA are pure $\mathcal{O}(n)$ make them more usable in practice.

7.4.1 From regex to NFA

Here is the list of regex when applied with NFA :

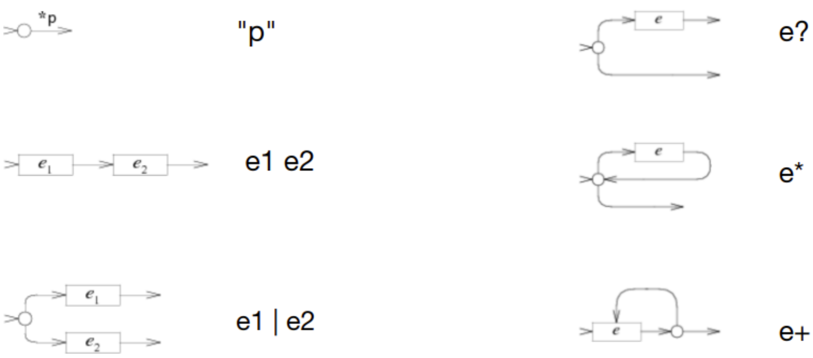

Figure 7.4: Regex to NFA

Note that e_1, e_2 are complete automaton that are plugged. Also if we don't have starting point (like in e_1e_2) it means that the starting state of e_1 become the starting state of the whole sequence.

Empty transitions are very useful in order to keep the number of transition manageable. Without them, every state would need to be link together (so, for 2 choice of 3 letters, 12 transitions, 9 with empty transitions.)

7.5 Powerset construction

Powerset construction is a technique that allow us to transform a NFA to DFA. It is quiet simple, we take concurrent equivalent states and merge them into a single one in the DFA. Figure 7.3 show the

7.6 Minimization of DFA

When we "translate" a NFA to a DFA we may end up with an increasing size! e.g a NFA of size n, may lead to an DFA of size 2^n. We cannot do anything for that expect to minimize a little the DFA.

These two algorithms are classical ones in compilers classes.

7.7 DFA simulation

It use the Thompson's algorithm :

- Run the NFA by maintaining a list of states
- Build the DFA by treating the list of states as a single DFA state.
- Need to be able to lookup a DFA state from a list of NFA state.

This approach is typically slower as we do the work while lexing. However, the main advantage is that we only build DFA node that we need!

Chapter 8

Pumping lemma

8.1 Recap

Definition 8.1.1 (Grammar)**.** Grammar = (N, Σ, P, s)

- N : Non-terminals
- Σ: Alphabet (terminals)
- P: Production rules
- Starting Symbol (S $\in$ N)
- $P : N \to \Sigma * N?$

Matchable by a DFA!

Regular languages are capable of expression repetition and optionality thanks to the optional final non-terminal. However, it can't do nesting.

We can ask ourself : how to know if a language is regular ? And how to prove (just build the grammar, is that match it works)/disprove (pumping lemma) it?

8.2 Central Recursion

Let's take an example of non-regular language: $A ::=' (' A')' | \epsilon$. This is called central recursion. Note that language with central recursion are never regular. That's why C(or Java) does not have nested comments

A way to check if a language is regular is to check if it is matchable by a DFA (that has only one piece of state, the current one). If we want to match the central recursion, we need an extra counter for the depth. It exists only one way to implement that : using non-terminal in the middle. The problem is : this is not allowed in regular languages. We can proof it using the pumping lemma.

8.3 The Pumping Lemma

Definition 8.3.1 (Pumping Lemma)**.** If L is a regular language:

- Then it exists an integer $p \geq 1$ specific to L
- Such that every sentence s in L of *length* $\geq p$ can be written $s = xyz$ satisfying the following conditions :
  - $|y| \geq 1$
  - $|xy| \leq p$
  - $\forall n \geq 0, xy^n z \in L$

Every long enough ($p \geq 1$ and specific to L) string in the language can be used to generate longer string through a repeating part (y).

Note that there is also a corollary : every long enough string can be decomposed into a prefix, a suffix and a repeating part.

Thanks to the pumping lemma, we can proof that $L = (^k)^k$ is not regular (using contraction). Indeed, as we can show that x and y have to be made only of '(' and that y cannot be empty by definition, if we try to repeat y, the parenthesis are unbalanced! Which is a contradiction.

The pumping lemma can show us that any finite language is regular! Even a really simple one like $L = a|b|c$. We just need to take a big enough p! (Like $p = 2$ in this case).

8.3.1 Why do we need $|xy| \leq p$?

If we omit it, we could prove that $L = (^k a+)^k$ to be regular which is obviously not the case!. We need central recursion to be bounded because of DFA state limit. The constraint ensures we're able to build a maximum depth counter-example.

8.3.2 Be care

Note the way the pumping lemma is written, it is an implication. That mean $A \implies B \neq B \implies A$. Being able to pump sentences does not make a language regular. However is it a necessary condition in order to be regular.

8.3.3 In CFGs

The pumping lemma also work in CFGs! The difference is instead of splitting a sentence in 3, we split it in 5 $s = vwxyz \implies vwwxyyz \in L$ the parenthesis language works if we pick w=(and y=) note that a language like $a^k b^k c^k$ does not work. In both case, the pumping lemma can prove it!

8.3.4 In everything else

We just have to decompose in a pattern :

- X be a set of sets (e.g regular languages)
- An infinite set (language) of objects (sentences) in X can be obtained if long object repeat some sub-elements
- We can use the pumping lemma to disprove the belonging of a language to X

Chapter 9

LL and LR algorithms

This chapter is interested in how to parse CFGs grammar. We will see two algorithms : LL and LR. Note that, they are not generic algorithms for CFGs parsing.

9.1 Historical perspective

- Chomsky's Hierarchy : mid-50s
- LL & LR : formalized in mid/late 60
- Processors at this time where not very efficient
- $\mathcal{O}(n^3)$ was laughable, way to slow

Taking that into account, the choice to make efficient implementation (no backtracking / fancy data structure) that can only parse a part of CFGs was made. It is very practical, indeed we can show that programming language typically can be parse in $\mathcal{O}(n)$.

9.2 LL

Definition 9.2.1 (LL)**.** LL = Left-to-right Leftmost derivation. It always expand the left non-terminal first.

It can parse a subset of CFGs :

- Choices: use k tokens of lookahead to decide (never backtrack)
- Nowadays we consider it as a worse PEG
- If the grammar is accepted by LL, it guarantees $\mathcal{O}(n)$, can be implemented based on table-based lookup

9.2.1 Properties

- Top down recursive descent algorithm as PEG (but only one choice alternative taken, depends on lookahead lookup)
- No left recursion allowed
- Like PEG : unambiguous

- No language hiding (
$$A ::= a * a$$
is not a valid language in LL)
- Less expressive than CFG or PEG (even than LR)

Definition 9.2.2 (LL Grammar)**.** Grammar for which we can generate an LL parser (no FIRST/FIRST or FIRST/FOLLOW conflicts)

Definition 9.2.3 (LL Language)**.** A language that has a LL grammar (may have multiple grammars including non-LL ones.)

LL Conflicts

FIRST/FIRST conflit

- Choices starting with the same k tokens
- $A ::= ab|ac(k = 1)$

FIRST/FOLLOW conflit

- Choice can start or (if it is nullable) be followed by the same k tokens ($S ::= XY; X ::= \epsilon|a; Y ::= a|b$)

We can avoid FIRST/FIRST with left-factoring (factor out the common part at the start of two choice alternatives)

• InClassDecl	::= FieldDecl | MethodDecl
FieldDecl	::= Modifier* Type Identifier ('=' Expression)? ';'
MethodDecl	::= Modifier* Type Identifier '(' ParameterList ')' Body
→	
InClassDecl	::= Modifier* Type Identifier (FieldDeclSuffix | MethDeclSuffix)
FieldDeclSuffix	::= ('=' Expression)? ';'
MethDeclSuffix	::= '(' ParameterList ')' Body

Figure 9.1: Left-factoring example

In the previous figure, the three first line are not LL as the two last starts with the same prefix. The 3 last lines are LL because the suffix has been extracted, so the both use the same rule with a suffix choice instead of two rules.

Note that it is not great for plain syntax tree, also makes the AST a bit more difficult to build. However it is also useful for PEG performance (we avoid parsing the same prefix many times).

9.2.2 LL(1) vs Regular Expression

As both of them use a single symbol we could ask why are they not the same.

- LL can handle central recursion
- Regular grammars are parser with $\mathcal{O}(1)$ space (current state)
- LL(1) implemented by top-down recursive descent uses $\mathcal{O}(n)$ space: the function call stack (can be use to recurse)

9.3 LR

LR algorithm is the most difficult parsing algorithm that we'll see. Still, it is not the most difficult one (hello GLR). The goal of LR is to parse every deterministic grammar. Every grammar can be parsed in $\mathcal{O}(n)$ without backtracking (we cannot explore different alternatives). Hence, it is not the same as LL(1) as LL(1) decides eagerly and ignores the context.

Definition 9.3.1 (LR parsing)**.** LR = Left-to-right Rightmost derivation. It always expand the right non-terminal first.

Essentially done by DPDA:

- Stack on which symbols can be pushed (terminal and non-terminals)
- Performs a table lookup based on the stack and some tokens of lookahead(LL(1)/LL(k))

9.3.1 Shift and reduce

The algorithm use a table that maps the stack to one of the two actions :

1. shift : the next terminal onto the stack
2. reduce : items at the top of the stack to a non-terminal.

If we take our JSON grammar, an example would be :

1. remaining input { "x": 1} Stack : []
2. Shift x4 times
3. Reduce to PAIR Stack : [{, "x", :, 1]
4. Reduce to TPAIRS(from ϵ) Stack : [{, PAIR]
5. Reduce to PAIRS Stack : [{, PAIR, TPAIRS]
6. Reduce to PAIRS? Stack : [{, PAIRS]
7. Shift Stack : [{, PAIRS? }]
8. Reduce to OBJECT Stack : [OBJECT] -> VALUE

9.3.2 LR Conflicts

REDUCE/REDUCE conflicts

- Rare case, it means the same set of symbols can be reduced to the same non-terminal, in the same context (same input prefix)
- Same portion of input could be matched to different non terminals (ambiguity)
- Non-trivial example can also appear using optional rules.

SHIFT/REDUCE conflicts A famous problem for this is

```
if (a) if (b) s1(); else s2();
```

The else can be interpreted as the false condition of the outer or of the inner if. Depending on that, the sequence of shift/reduce will not be the same. That's why parsing tool let us define instruction for that (common default behavior is to prioritize shift over reduce).

9.3.3 Ambiguity

Definition 9.3.2 (Ambiguity)**.** Ambiguity appear when there are multiple way to parse the same input :

- Different (combination of) rules can match the same input
- Simulating different derivations
- Generating different plain parse trees

It is undecidable for CFGs. Note that deterministic grammars are unambiguous. Determinism is decidable (non-determinism = conflict). Thus, ambiguity $\implies$ Non-determinism, by modus tollens Determinism $\implies$ Unambiguity.

However, it exists unambiguous, non-deterministic grammars (palindrome language).

In the end, what is not LR ?

- Ambiguous grammars
- Corner-case speculative scenarios like above (rare case)

9.3.4 LR building table

This is the most difficult part of LR, but it is not needed to be able to use LR and its variants.

9.3.5 Variants

- LR(k): use more lookahead
- LALR: lose some context given by the prefix (smaller tables than LR)
- SLR: lose all context given by prefix (even smaller tables)
- IELR: True LR, tables as small as LALR
- GLR: can parse every CFG in $\mathcal{O}(n^3)$, deterministic ones in $\mathcal{O}(n)$

See some examples of LALR, SLR and LR(k) in the slides.

GLR

GLR is used when we need to speculate/backtrack we fork the LR stack. It uses graph-structured stacks (GSS). It shares as much of the stack as possible with the possibility to merge stacks down the line.

It is a non-deterministic pushdown automaton.

9.4 Other variants for CFG

Some other parsers can parse every CFG grammars.

- Early (as seen in Computational Linguistic course)
  - $\mathcal{O}(n^3)$
  - Popular for Linguistic (lots of ambiguity)
  - Simplest general parsing algorithm

- ANTLR/ALL(*)
  - $\mathcal{O}(n^4)$
  - Very popular java parsing tool
  - LL(k) + backtracking + caching via automata
- GLL
  - $\mathcal{O}(n^3)$
  - LL + GSS

Chapter 10

Semantic Analysis

With semantic analysis we begin the next step of our pipeline. We begin the semantic analysis with an AST.

10.1 Introduction

Semantic analysis allow us to check everything that :

- Can be check statically (without running the program) in reasonable time (no real symbolic execution (instead of real input, we use variable. The problem is when we encounter if branching because it multiply the amount of path (exponential) this phenomenon is called combinatorial explosion))
- That we didn't check in the parser (we should check as little as possible in the parser)

10.1.1 Check after parsing

We can take the example of Java modifiers if we would like to check that in the parser we would need many rules, for each modifiers, the methods without them, etc. Thus, it would only be for non-abstract classes as the other (like interface) have other constraints. Besides, it does not prevent thing like "protected static" which is forbidden in Java.

A first better way is to use Autumn parsing combinator, but still we should check that with the semantic analysis. An example of error for the following : public private void test()[] would be :

- Parser : "unexpected token 'private'"
- Semantic: "two visibility modifiers for method"

No need to say that the second one is much better. Also, incorrect ASTs can be use in IDEs (syntax highlighting, etc.)

10.1.2 Main Concerns

Type checking

Definition 10.1.1 (Type checking)**.** Check type constraints : "int x = "String";" but also type inference : "var x = "string" + 42. If the language is dynamically typed it is done during runtime and libraries.

Name Binding

Definition 10.1.2 (Name binding)**.** Check where name are defined. e.g : "int x = y + 3;" What is y? Inter-dependency : "var x = a.b.c" we need to know the type of c, that need the type of b, that need the type of a.

These two first principle cannot be done separate.

Note that Name binding also consist of lexical scoping (what is visible or not for a field/inside a method/block etc.). Dynamic scoping can also be done, however is it opposed with lexical scoping. Still it have this advantages! Emacs use this kind of scoping, this allow Emacs to edit Ruby and Python file and handle tabulation in both case.

Flow checks

Flow checks is a complex think, let's take a Java example :

```java
int test(int x) {
    if (x == 3) return 3;
}
int test2(int x) {
    while(true) return 1337;
}
```

The first one is an invalid Java program (suppose to return an int but only return it if x==3) and flow checks spot that. The second one is valid because Java detects that the while loop will always be entered.

More...

Languages like Whiley / Dafny / Rust that statically does thinks.

10.2 The Hindley-Milner Type System

10.2.1 Introduction

Type system for polymorphic lambda calculus (System F). It is used in practice as it is the base of Haskell and ML type system. Also it can be extended in various ways.

10.2.2 Lambda calculus

Lambda calculus

Toy language that is written this way : $e ::= x|(\lambda x.e)|(ee)$ variable/Abstraction/application

Simply-Typed Lambda calculus

$e ::= x|(\lambda x : \tau.e)|(ee)|c$ where τ us the parameter type and c a constant. Note that, we need a set of base types e.g B = {a, b}. We do need constant c, because unlike classical lambda calculus we do not have the Church encoding (define number as function).

Polymorphic lambda calculus

The idea is to type the original lambda calculus without the type annotations
$e ::= x|(\lambda x.e)|(ee)|let x = e \text{ in } e$

10.2.3 Type system

Definition 10.2.1 (Type system)**.** A formal system(formalism) that determines :

- if any expression in the language is well-typed
- type of any such expression
- $e ::= x|(\lambda x : \tau.e)|(ee)|c$
- ill-typed: $((\lambda x : a.x)c1)$ where c1 has type b

We would like the type system to be :

- **Decidability** : can make a decision
- **Soundness** : everything we can prove is true
- **Completeness** : we can prove everything that is true

However the Gödel's incompleteness theorem says that "No sound system of axioms describing natural number arithmetic can be complete". Still, if we look in practice, Java does not have any of the previous properties...

For λ

- Γ = context, a set of type bindings (assignments, assumptions)
- ⊢ = type judgement
- σ, τ denote types

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x{:}\sigma}\ (1) \qquad \frac{c \text{ is a constant of type } T}{\Gamma \vdash c{:}T}\ (2)$$

$$\frac{\Gamma, x{:}\sigma \vdash e{:}\tau}{\Gamma \vdash (\lambda x{:}\sigma.\ e){:}(\sigma \to \tau)}\ (3) \qquad \frac{\Gamma \vdash e_1{:}\sigma \to \tau \quad \Gamma \vdash e_2{:}\sigma}{\Gamma \vdash e_1\ e_2{:}\tau}\ (4) \qquad ((\lambda\ x{:}a\ .\ x)\ c)$$

Figure 10.1: Type system for λ

10.2.4 Towards Hindley-Milner

The key idea is : function can have many types (polymorphism). Why is it important? Because we really need polymorphism in order to be able to write rule like $((\lambda id.((foo(id1))(id's')))(\lambda x.x))$ without polymorphism, the identity function that we use should be declared as integer or char with polymorphism it can return both!

Generalization

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_D x : \sigma}\ [\texttt{Var}]$$

$$\frac{\Gamma \vdash_D e_0 : \tau \rightarrow \tau' \qquad \Gamma \vdash_D e_1 : \tau}{\Gamma \vdash_D e_0\ e_1 : \tau'}\ [\texttt{App}]$$

$$\frac{\Gamma,\ x : \tau \vdash_D e : \tau'}{\Gamma \vdash_D \lambda\, x\, .\, e : \tau \rightarrow \tau'}\ [\texttt{Abs}]$$

```
⊢ (λ x . x) : α → α           (Abs)
⊢ (λ x . x) : ∀ α . α → α     (Gen)
```

$$\frac{\Gamma \vdash_D e : \sigma \quad \alpha \notin \mathrm{free}(\Gamma)}{\Gamma \vdash_D e : \forall\, \alpha\, .\, \sigma}\ [\texttt{Gen}]$$

Which are variables are bound (unfree)?

- ∀-quantified variables
- variables appearing in function types
  - ... that never appear on their own

Figure 10.2: Generalization

Still, it this step we cannot apply the function as we still have the polymorphic type and not a specialized type.

Intantiation

Allow us to specify a type from polymorphic type!

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_D x : \sigma}\ [\texttt{Var}]$$

$$\frac{\Gamma \vdash_D e_0 : \tau \rightarrow \tau' \qquad \Gamma \vdash_D e_1 : \tau}{\Gamma \vdash_D e_0\ e_1 : \tau'}\ [\texttt{App}]$$

$$\frac{\Gamma,\ x : \tau \vdash_D e : \tau'}{\Gamma \vdash_D \lambda\, x\, .\, e : \tau \rightarrow \tau'}\ [\texttt{Abs}]$$

```
⊢ (λ x . x) : α → α           (Abs)
⊢ (λ x . x) : ∀ α . α → α     (Gen)
⊢ (λ x . x) : int → int       (Inst)
```

$$\frac{\Gamma \vdash_D e : \sigma \quad \alpha \notin \mathrm{free}(\Gamma)}{\Gamma \vdash_D e : \forall\, \alpha\, .\, \sigma}\ [\texttt{Gen}]$$

$$\frac{\Gamma \vdash_D e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash_D e : \sigma}\ [\texttt{Inst}]$$

Why do we need polymorphism?

```
((λid . ((foo (id 1)) (id 's')))
 (λx . x))
```

Figure 10.3: Instantiation

Let polymorphism

The let rule allow us to go to a monomorphic type!

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_D x : \sigma} \quad [\texttt{Var}]$$

$$\frac{\Gamma \vdash_D e_0 : \tau \rightarrow \tau' \qquad \Gamma \vdash_D e_1 : \tau}{\Gamma \vdash_D e_0\ e_1 : \tau'} \quad [\texttt{App}]$$

$$\frac{\Gamma,\ x : \tau \vdash_D e : \tau'}{\Gamma \vdash_D \lambda\, x\ .\ e : \tau \rightarrow \tau'} \quad [\texttt{Abs}]$$

$$\frac{\Gamma \vdash_D e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash_D e : \forall\, \alpha\ .\ \sigma} \quad [\texttt{Gen}]$$

$$\frac{\Gamma \vdash_D e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash_D e : \sigma} \quad [\texttt{Inst}]$$

$$\frac{\Gamma \vdash_D e_0 : \sigma \qquad \Gamma,\ x : \sigma \vdash_D e_1 : \tau}{\Gamma \vdash_D \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : \tau} \quad [\texttt{Let}]$$

Figure 10.4: Let polymorphism

We just need to rewrite it like : $letid = (\lambda x.x)in((foo(id1))(id's'))$

10.2.5 Type system vs Typing algorithm

Thanks to inference rules we have the semantics of the type system. Rules allow us to prove statements about types, as the system is sound the created statements are true. However, the way to prove statements is not given (algorithm is needed for that).

10.3 Using Uranium