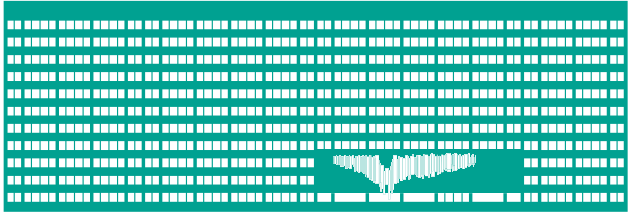
VŠB TECHNICKÁ UNIVERZITA OSTRAVA

VSB TECHNICAL UNIVERSITY OF OSTRAVA



www.vsb.cz

Programming Languages and Compilers

behalek.cs.vsb.cz/wiki/index.php/Programming_Languages_and_Compilers

Marek Běhálek

VSB – Technical University of Ostrava

marek.behalek@vsb.cz

March 2, 2022

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

1. Introduction to the area of programming languages and compilers
2. History of programming languages
   - First programming languages
   - Notable Programming languages
3. Classification of Programming Languages
4. Specification of programming languages
5. Compilers
   - Basic properties
   - Types of compiles
   - Transformation of Source Codes
   - Compiler's inner structure
6. Formal languages
7. Chomsky Hierarchy
8. Lexical analysis
9. Syntactic analysis
   - LL(1) Grammars
10. Implementing Parser for LL(1) Grammars
    - Recursive Descent
    - Non-recursive Predicative Analysis for LL(1)
    - Compiler-Compilers
11. LR Grammars
12. Error Handling

Motivation

- Get a better insight into programming languages. We will talk about the classification of programming languages.
- Understand a structure of a compiler and basic compiler's types.
- We will learn how to build a parser based on LL(1) grammars.
- We will learn how to use *compiler compilers*, namely ANTLR.
- We will talk (briefly) about other aspects like optimizations and run-time environment.
- Building a real programming language compiler is rare, but task like extracting data from source codes, parsing some protocols or converting data formats are pretty common $\rightarrow$ the same techniques can be used $\rightarrow$ much simpler then most ad-hoc solutions

What is a programming language?

- Plenty of *informal definitions*.
  - A programming language is a machine-readable artificial language designed to express computations that can be performed by a machine, particularly a computer.
  - Programming languages can be used to create programs that specify the behavior of a machine, to express algorithms precisely, or as a mode of human communication.
- Frequently, the term *programming language* is restricted to those languages that can express all possible algorithms.
  - Turing complete languages
  - SQL or HTML are languages (artificial, machine-readable, expressing computation, ...), that are usually **not** considered programming languages.
- If we want to use (especially high-level) programming languages we need *a compiler*.
  - A product of this compiler is a computer program - a collection of instructions that perform a specific task when executed by a computer.

First programming languages

- Theoretical beginnings (early 20^{th} century)
  - Alonzo Church - lambda calculus – theory of computations
  - Alan Turing – show that a machine can solve a "problem".
  - John von Neumann – defined computer's architecture (relevant even for today's computers).
- First programming languages (Wikipedia)
  - **Plankalkül** - 1942 - 1945, Konrad Zuse, used for *mechanical* computer Z1.
  - **Short Code** - early 1950s, John Mauchly, first functioning programming languages designed to communicate instructions to an electronic computer.
  - **FORTRAN** - 1954 - 1957, IBM, John Backus, first high-level general purpose programming language to have a *working* implementation.
    - Even now, along with C/C++, most frequently used programming language for high-performance computing.

First programming language - FORTRAN

- FORTRAN compiler - biggest *programming* project of that times.
- More practical alternative to assembly language (still we are talking about punched cards era).
  - Even assembly language evolved, now is much more *higher level and user friendly*.
- Many versions:
  - FORTRAN II (1958, procedural programming), III (1958), IV (1961, booleans)
  - FORTRAN 66, FORTRAN 77 (structured programming)
  - Fortran 90 (Major revision, close to C), Fortran 95 (included High Performance Fortran)
  - Modern fortran: Fortran 2003 (OOP,streams, ...), Fortran 2008, Fortran 2018

First programming language - FORTRAN II - One card input

```
C AREA OF A TRIANGLE - HERON'S FORMULA
C INPUT - CARD READER UNIT 5, INTEGER INPUT
C OUTPUT -
C INTEGER VARIABLES START WITH I,J,K,L,M OR N
      READ(5,501) IA,IB,IC
  501 FORMAT(3I5)
      IF (IA) 701, 777, 701
  701 IF (IB) 702, 777, 702
  702 IF (IC) 703, 777, 703
  777 STOP 1
  703 S = (IA + IB + IC) / 2.0
      AREA = SQRT( S * (S - IA) * (S - IB) * (S - IC) )
      WRITE(6,801) IA,IB,IC,AREA
  801 FORMAT(4H A= ,I5,5H  B= ,I5,5H  C= ,I5,8H  AREA= ,F10.2, $13H SQUARE UNITS)
      STOP
      END
```

First programming languages - Fortran 77

```
*     Prints the values of e ** (j * i * pi / 4) for i = 0, 1, 2, ..., 7
*         where j is the imaginary number sqrt(-1)

      PROGRAM CMPLXD
        IMPLICIT COMPLEX(X)
        PARAMETER (PI = 3.141592653589793, XJ = (0, 1))
        DO 1, I = 0, 7
          X = EXP(XJ * I * PI / 4)
          IF (AIMAG(X).LT.0) THEN
            PRINT 2, 'e**(j*', I, '*pi/4) = ', REAL(X), ' - j',-AIMAG(X)
          ELSE
            PRINT 2, 'e**(j*', I, '*pi/4) = ', REAL(X), ' + j', AIMAG(X)
          END IF
    2     FORMAT (A, I1, A, F10.7, A, F9.7)
    1     CONTINUE
        STOP
      END
```

First programming languages - Modern fortran (Fortran 90 +)

```
program area
    implicit none
    real :: A, B, C, S

    ! area of a triangle
    read *, A, B, C
    S = (A + B + C)/2
    A = sqrt(S*(S-A)*(S-B)*(S-C))
    print *,"area =",A
    stop
end program area
```

```
program factorial
   integer :: i, num
   integer (kind=16) :: fact

   write (*,*) "n! ? "
   read(*,*) num

   fact = 1
   do i = 2, num
      fact = fact * i
      write(*,'(I3,"! = ", I20)') i, fact
   end do
end program factorial
```

Historical *Influencers*

LISP - (second oldest) 1958, John McCarthy (MIT)

- Originally mathematical notation for computer programs.
- Functional programming style.
- Pioneered many (now widely used) ideas (tree data structures, high order functions, recursion, dynamic typing,...).
- Many dialects: Common Lisp, Scheme, Clojure

```
(defun factorial (n)
    (if (zerop n) 1
        (* n (factorial (1- n)))))
```

Historical *Influencers* II

- **Algol** - 1958-1960, Bauer, Bottenbruch, Rutishauser, Samelson, Backus, Katz, Perlis, Wegstein, Naur, Vauquois, van Wijngaarden, Woodger, Green, McCarthy
  - Notable versions: Algol 60, Algol 68
  - Heavily influenced many other programming languages (especially syntax) - C/C++, Pascal, Simula, Smaltalk, Ada, ...
  - ACM - standard method for algorithm description in academia text books for decades.
  - Algol 68 had garbage collection.
- **COBOL (common business-oriented language)** - 1959, CODYSAL
  - It is an imperative, procedural and, since 2002, object-oriented language.
  - Primarily used in business, finance, and administrative systems.
  - Most programming in COBOL is now purely to maintain existing applications.
  - English-like syntax.

Historical *Influencers* II - Algol 60

```
procedure Absmax(a) Size:(n, m) Result:(y) Subscripts:(i, k);
    value n, m; array a; integer n, m, i, k; real y;
comment The absolute greatest element of the matrix a, of size n by m,
    is transferred to y, and the subscripts of this element to i and k;
begin
    integer p, q;
    y := 0; i := k := 1;
    for p := 1 step 1 until n do
        for q := 1 step 1 until m do
            if abs(a[p, q]) > y then
                begin y := abs(a[p, q]);
                    i := p; k := q
                end
end Absmax
```

Historical *Influencers* II - COBOL

```
COMPUTE-GROSS-PAY.
       IF HOURS-WORKED > 40 THEN
           MULTIPLY PAY-RATE BY 1.5 GIVING OVERTIME-RATE
           MOVE 40 TO REGULAR-HOURS
           SUBTRACT 40 FROM HOURS-WORKED GIVING OVERTIME-HOURS
           MULTIPLY REGULAR-HOURS BY PAY-RATE GIVING REGULAR-PAY
           MULTIPLY OVERTIME-HOURS BY OVERTIME-RATE
               GIVING OVERTIME-PAY
           ADD REGULAR-PAY TO OVERTIME-PAY GIVING GROSS-PAY
       ELSE
           MULTIPLY HOURS-WORKED BY PAY-RATE GIVING GROSS-PAY
       END-IF
       .
```

Some popular languages

- Structured programming languages
  - **Pascal** - 1968-71, Niklaus Wirth, ETH Zurich
  - **C** - 1972, Dennis Ritchie, Bell Labs
- Object Oriented Languages
  - **Simula 67** - 1960s, Ole-Johan Dahl, Kristen Nygaard
  - **Smalltalk** - first version 1972, Alan Kay, Xerox, public version 1980 (Smalltalk-80),
  - **C++** - 1982-85, Bjarne Stroustrup, AT&T Bell Labs
- Statically typed functional programming
  - **ML** - 1973, Robin Milner, University of Edinburgh
- Logic programming
  - **Prolog** - 1972, Alain Colmerauer, Robert Kowalski
- *Popular* programming languages
  - **Visual basic** - 1991, Microsoft
  - **Python** - 1991, Guido van Rossum
  - **Javascript** - 1995, Brendan Eich, Netscape
  - **PHP** - 1995, Rasmus Lerdorf
  - **Java** - 1995, James Gosling
  - **C#** - 2000, Anders Hejlsberg, Mads Torgersen, Microsoft
  - Matlab, R, Ruby, Objective-C, Rust, Ada, Scala, Haskell, Lua, Rust, Go, Erlang,...

Programming languages and computer's architecture

- Programming languages are limited and affected by an architecture of computers.
  - Effective implementation must exists, if we want to use them to create real life applications.
  - Still true, but smaller problem then computer's stone age (1960s-1970s).
  - When computer's architecture is changing, the programming languages are also affected.
- Von Neumann's architecture (Princeton architecture) $\rightarrow$ very old (1945), but still *(somehow)* valid.
  - Architecture x86 started with Intel 8086 in 1978.
- Plenty of languages have the same basic constructs (Java, C++, C#, Python,...) like loops and conditions $\rightarrow$ these can be easily converted to current CPUs machine code.
- The approach: *from architecture to programming language* was criticized even in these stone ages.
  - Backus J. *Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs*, Communications of the ACMVolume 21Issue 8Aug. 1978 pp 613–641https://doi.org/10.1145/359576.359579 (Turing Award)

Classification of Programming Languages I

- How can we divide programming languages into categories? What are the most notable features? What should I list when describing programming language?
- A level of programming language (sometimes the term *generation* is used)
  - Low-level programming languages
    - Machine Code (1^{st} generation)
    - Assembly language (2^{nd} generation)
  - High-level programming languages (3^{nd} generation)
    - Majority of today's programming languages: C/C++, Java, Python,...
    - Requires *a compiler* to work.
    - Instead of basic elements like registers or instructions we use more complex ones like object or loops.
  - 4^{th}generation languages tend to be specialized toward very specific programming domains (database, GUI, Web).

Classification of Programming Languages II

- 5^{th} generation languages are based on problem-solving using constraints given to the program, rather than using an algorithm written by a programmer (constraint or logic programming).

- Programming paradigm - classify programming languages based on their features or a *style* (sometimes loosely defined).
- Notable features for their *implementation*.
  - Type system, Compiler design, Execution strategies, ...
- Programming domain (domain specific programming languages) - scripting, expert systems, artificial intelligence, natural language processing, numerical mathematics (statistics), ...

Machine Code

- Lowest level of programming.
- Consisting of machine language instructions, which are used to control CPU.
- Each instruction causes the CPU to perform a very specific task, such as a load, a store, a jump, or an arithmetic logic unit (ALU) operation on one or more units of data in the CPU's registers or memory.

```
[  op   |   rs  |   rt  | address/immediate]
   35       3       8             68            decimal
 100011  00011  01000  00000  00001  000100      binary
```

Assembly languages

- The same *level* of programming as machine code, instruction are in more human readable form.
- Still, they can be transformed to machine code using some sort of *macros*.
- Typically, one machine instruction is represented as one line of assembly code.

- Some high level languages (for example C) allows to use directly Assembly language code fragments.

```
_fib:
        movl $1, %eax
        xorl %ebx, %ebx
.fib_loop:
        cmpl $1, %edi
        jbe .fib_done
        movl %eax, %ecx
        addl %ebx, %eax
        movl %ecx, %ebx
        subl $1, %edi
        jmp .fib_loop
.fib_done:
        ret
```

Logic programming

- Largely based on formal logic.
- The idea behind: Algorithm = Logic + Control (different theorem-proving strategies)
- A program is then a set of sentences in logical form, expressing facts and rules about some problem domain.
- One of the most known logic programming language is Prolog
  - Often used in the area of artificial intelligence and computational linguistics.

```
edge(a, b). edge(a, c). edge(b, d). edge(c, d). edge(d, e). edge(f, g).

connected(N, N).
connected(N1, N2) :- edge(N1, L), connected(L, N2).
```

Programming paradigms I

- Programming paradigms defines a sort of *style* of programming.
  - Sometimes, it defines what is not allowed in particular style of programming (side effects in functional programming).
  - Programming paradigms are a way to classify programming languages based on their features.
- One language can implement more than one programming paradigms.
- Frequently, they implement just some selected parts of programming paradigms, *pure* languages are rarer.
- Some programming paradigms works well together or complements each other (structured and imperative programming), some are conflicting with each other (declarative and imperative programming).
- Most of modern programming languages are multi-paradigm languages (they implement several programming paradigms).

Programming paradigms II

- Popular programming paradigms
  - Imperative vs. Declarative (functional, logic) programming
  - Object oriented programming (class based, prototype based)
- But there are many more...
  - Visual programming, Event driven programming, Flow driven programming, Constraint programming, Aspect oriented programming, ...

Type system I

- **Type checking** - the process of verifying and enforcing the constraints of types.
  - Static typing *(C, C++, Java, Haskell...)*
    - Type checking is performed (mostly) during compile-time.
    - Static typing is a limited form of program verification.
    - It allows many errors to be caught early in the development cycle.
    - Static type checkers are conservative - they will reject some programs that may be well-behaved at run-time, but that cannot be statically determined to be well-typed.
  - Dynamic typing *(Javascript, Python, PHP...)*
    - Majority of its type checking is performed at run-time.
    - Dynamic typing can be more flexible than static typing. For example by allowing programs to generate types based on run-time data.
    - Run-time checks can potentially be more sophisticated, since they can use dynamic information as well as any information that was present during compilation.

```
var x := 5;      // (1)  (x is an integer)
var y := "37";   // (2)  (y is a string)
var z := x + y; // (3)  (? - Visual Basic = 42, Javascript "537")
```

Type system II

- Strongly typed languages - do not allow undefined operations to occur.
- Weak typing means that a language implicitly converts (or casts) types when used.
- Type safe - is language if it does not allow operations or conversions which lead to erroneous conditions.
- Memory safe - for example it will check array bounds (resulting to compile-time and perhaps run-time errors).

```
int x = 5;
char y[] = "37";
char* z = x + y; //z points five characters after y
```

- Polymorphism
  - The ability of code (in particular functions, methods or classes) to act on values of multiple types, or the ability of different instances of the same data-structure to contain elements of different types.
  - Type systems that allow polymorphism generally do so in order to improve the potential for code re-use.

Type system III

```
Animal obj = new Horse();
obj.sound();
length :: [a] -> Int -- a is a type variable
length [] = 0
length (x:xs) = 1 + length xs
```

- **Type interference**
  - Strongly statically typed languages
  - Automatic deduction of the data types
  - Hindley-Milner type system
- Other type systems categories
  - *Manifest* vs. *Inferred*
  - *Nominal* (C#, Java, - type are compared based on its definition trough name) vs. *Structural* (OCalm - types are compared based on their structure)

What we want to describe? I

- How a correct program looks like? $\rightarrow$ **Syntax**
  - Programming language syntax is generally distinguished into three levels:
    - Words – the lexical level, determining how characters form *tokens*;
    - Phrases – the grammar level, narrowly speaking, determining how tokens form phrases;
    - Context – determining what objects or variables names refer to, if types are valid, etc.
  - How to do that? $\rightarrow$ Formal languages, grammars, automatons,...
  - Backus–Naur form
    - It is a meta-syntax notation for context-free grammars.
    - Often used to define a syntax for programming languages.
    - There are some extensions. For example Extended BNF allows to use regular expressions * operator.

What we want to describe? II

```
<expression>  ::= <expression> + <term>
                | <expression> - <term>
                | <term>
<term>        ::= <term> * <factor>
                | <term> / <factor>
                | <factor>
<factor>      ::= <element> ^ <factor>
                | <element>
<element>     ::= ( <expression> )
                | <variable>
                | <number>
```

- What a correct program should do? $\longrightarrow$ **Senamtics**
  - Semantics reflects the meaning of programs. It is defined by some sort of *model*.
  - Many different frameworks, none of them considered to be *a standard*.
  - Much harder (than syntax definition) for most languages $\longrightarrow$ no formal definition for *most* languages.

What we want to describe? III

- Easier for languages that are closer to math, like functional languages. Usually it is formally defined for *a core*, other language constructs are then transformed into this core elements.
- Main approaches belong to three major classes:
  - *Axiomatic semantics* - meaning of phrases is described by axioms that apply to them. Example: Hoare logic.
  - *Operational semantics* - the execution of the language is described directly, this may be defined using an abstract machine (transitions on its state) or via syntactic transformations on phrases of the language itself.

$$\frac{< b, \sigma > \Downarrow \mathtt{true} \quad < c_0, \sigma > \Downarrow \sigma'}{< \mathtt{if\ b\ then\ c_0\ else\ c_1}, \sigma > \Downarrow \sigma'} \qquad \frac{< b, \sigma > \Downarrow \mathtt{false} \quad < c_1, \sigma > \Downarrow \sigma'}{< \mathtt{if\ b\ then\ c_0\ else\ c_1}, \sigma > \Downarrow \sigma'}$$

  - *Denotation semantics* - meanings are modeled by mathematical objects that represent the effect of executing this construct. Thus only the effect is of interest, not how it is obtained.

What we want to describe? IV

```
data Com = If    Exp Com Com  -- if( e ) c else c
         | While Exp Com      -- while( e ) c
         | Seq   Com Com      -- c ; c
data State  = State {store::Store, input::[Int], output::[Int]}

p :: Prog -> Input -> Output
p body inp = out
             where State{output=out} = c body (initialState inp)

c :: Com -> State -> State
c (If b c1 c2) s = let (v1,s') = e b s
                   in if v1 == 0 then c c2 s'
                      else c c1 s'
c (While b c1) s = let (v1,s') = e b s
                   in if v1 == 0 then s'
                    else c (While b c1) (c c1 s')
c (Seq c1 c2) s = c c2 (c c1 s)
```

What is a compiler?

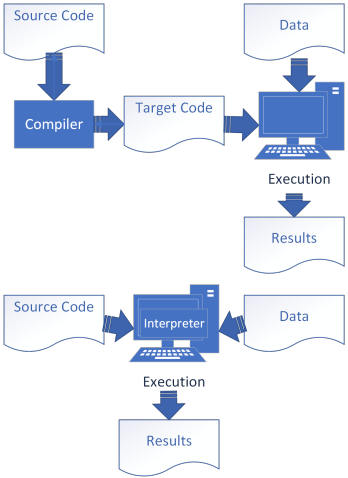
- We all agree, some kind of *a compiler* is necessary, when we want to use a high level language.
- **Compiler's primary function is to translate a source language to a target language.**
  - Most often, the source language is a high-level programming language *readable* by humans and the target is then a low-level language *understandable* by a computer.
  - Still there are other types of *compilers* that for example: extracts some data from source codes (Doxygen), generates documents from source codes (Latex), works with data (XSLT), ...
  - Possible are also source to source compilers, that transform one program to another (OpenMP).
- Are there some secondary requirements?
  - Handle errors - How and when to report errors?
  - Optimizations - Most often, it is not enough to produce just some working result.
  - Supporting activities like debugging, profiling,..

Most common scenario for source and target languages

- Source language
  - Most often programming languages (C, Java, Haskell,...).
  - Special (domain oriented) languages (Latex, VHDL, PostScript,...).
- Target language
  - Machine code
  - Object code - most often *machine code modules* that requires a linker to produce an executable or library file.
  - Bytecode, intermediate language, p-code (portable code) - assembly code or machine code for a virtual machine.
  - Less frequent can be generating assembly code, or some high level language (C for embeded systems).

Types of Compiles I - *Traditional* Compiler vs. Interpreter

- Compiled vs. Interpreted languages - usually not strictly given, sometime one type is better suited.
- Compiler advantages:
  - **Faster**, usually better optimizations
  - It is not require in a run-time.
  - Best suited for applications like HPC.
- Interpreter advantages:
  - **Less platform dependent**
  - Better suited for run-time modifications.
  - Best suited for scripting or web languages.
  - Often simpler - better for education.



Types of Compiles II

- Native code compiler vs. Cross Compile (a program that translates into an object code format that is not supported on the compilation machine. Suited for embedded applications).
- Ahead-Of-Time (AOT) compiler - program is compiler to machine code before its execution (usually emphasize some kind of advantages from the act, while a traditional compiler works technically like this).
- Just-In-Time (JIT) compiler = AOT compilation + interpretation
  - JIT compiler performs compilation to machine code during execution of a program.
  - Tries to combine main advantages of *a compiler and an interpreter* (also combines some disadvantages of both approaches).
  - *Frequently used now-days.*
- Decompiler, Disassembler - translates *an executable codes* to high-level source codes.
- Parallelizing Compiler, Bootstrap Compilers, Incremental Compiler, Source-To-...

Transformation of Source Codes I

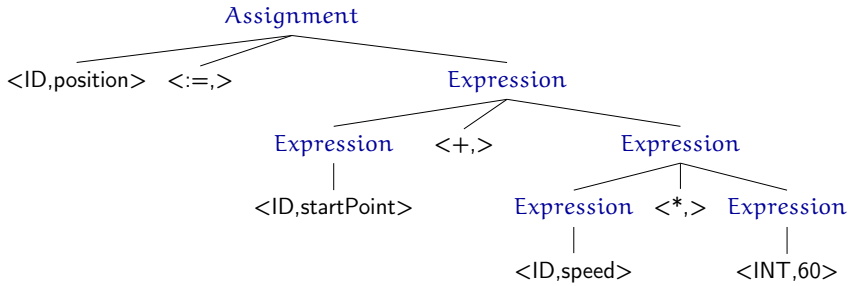
- Source code - input is a text at this stage.

```
position := startPoint + speed * 60;
```

- Lexical analysis - result is a sequence of tokens (lexical symbols).

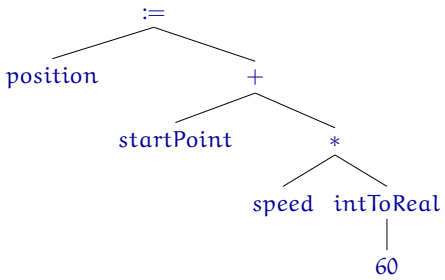
```
<ID,position> <:=,> <ID,startPoint> <+,> <ID,speed> <*,> <INT,60>
```

- Syntactic analysis, parsing - defines a hierarchy (corresponding parenthesis are matched), the result us usually Abstract Syntax Tree (AST).



Transformation of Source Codes II

- Semantic (context sensitive) analysis
  - Solves issues that are hard (impossible) to capture by context free languages.
  - Adds semantic information to the parse tree.
  - It usually includes type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use).



Transformation of Source Codes III

- Transformation to some kind of intermediate representation.
  - Different types of representation, the choice is strongly dependent on operations we want to perform.

```
temp1 := inttoreal(60)
temp2 := speed * temp1
temp3 := startPoint + temp2
position := temp3
```

- Optimization

```
temp1 := speed * 60.0
position := startPoint + temp1
```

- Target code generation

```
fld  qword ptr [_speed]
fmul dword ptr [00B2]                ;60.0
fadd qword ptr [_startPoint]
fstp qword ptr [_position]
```

Compiler's inner structure I

Compiler's inner structure can be described from different perspectives.

1. We can decompose the compilation into *logical* phases.
   - They roughly follows how the language models are transformed during the compilation process.
   - Then, the structure of a compiler can be described as units (corresponding to these logical phases) collaborating (along with *supporting units* like a symbol table) to achieve desired result.
   - These logical phases can be divided into two (three) stages: *front end,* *(middle end,)* *back end*.
   - Front-end (analysis)
     - Preprocessing, lexical analysis, syntactic analysis (parsing), semantic analysis (syntax-directed translation),
     - The result of this stage is some kind of intermediate representation (usually lower level, more suitable for further stages).
   - Middle-end (optimizations)

Compiler's inner structure II

- Particular operations usually compose from two steps: analysis (data-flow analysis, dependence analysis, alias analysis, pointer analysis, escape analysis,...) and the optimization itself (inline expansion, dead code elimination, constant propagation, loop transformation, automatic parallelization,...).
- These optimizations are often machine and even source code independent.
- Optimizations modifies used intermediate representation, or generate a new, more appropriate one.

- Back-end (synthesis)
  - Target code is generated, this stage is target hardware architecture dependent.
  - Machine code optimizations, code generation, target code optimizations.

2. We can classify compilers by number of passes they do.
   - One pass can be roughly described as a process, when the compiler *goes trough* source codes (or their representation).
   - Funny history of multi-pass compilers
     1. First compilers were multi-pass compilers, due to hardware limitations.
     2. One-pass compilers were harder to implement, but were *better* mainly due to being faster.

Compiler's inner structure III

3. Languages were designed to allow one-pass compilation (C,Pascal).
4. Hardware limitations are long gone, multi-pass compilers are able to perform better optimizations, thus producing better quality target code.
5. Modern programming languages (like Java) have *simpler* design, but require multi-pass compilers.
6. Most modern compilers are multi-pass again...

- One-pass compiler - all logical phases are performed in one pass.
  - Limited optimizations, but faster $\rightarrow$ useful in some areas like scripting languages or development tools.
  - Simpler to handle errors $\rightarrow$ useful for education.
- Muti-pass compiler (optimizing compiler)
  - Specific optimizations are usually performed in *one pass*, levels of optimizations can be defined by a programmer.
  - Is able to produce better target code, but are more complex.

Compilers re-usability

Compiler's inner structure IV

- Because of these distinct logical phases and stages, real compiler's design usually compose from modules, with clear purpose $\rightarrow$ we do not need to start from scratch $\rightarrow$ we can reuse existing solution.
- One of real open source project, providing robust compilers infrastructure (that can be extended in many ways) is: LLVM `https://llvm.org/`

Sometimes, real compilers uses separated tool like: pre-processor, assembler or linker.

Creating a compiler

Historical compilers (for example for FORTRAN) were biggest software project of that time. Now-days, similar project is much simpler. Key question is why?

- There are no hardware obstacles as before. Software engineering is much more advanced.
- We can reuse *other* compilers.
- There are plenty of special supportive tools. Most notably compiler compilers and compiler infrastructure projects like LLVN.
- Especially the analysis part is well established - backed by *theory*, mainly formal languages.

Formal languages - simplified

- Alphabet - finite set of symbols Σ
  - Example: {0,1}, {a, b, c, ... z}, {a,b,+, -,(,)}
- Words over an alphabet Σ - denoted by Σ^*
  - Set of symbols from Σ + Empty word ε
  - Example: 1001, pjp, a-(b+b)
- Language over an alphabet Σ
  - A subset of words (Σ^*) over an alphabet (Σ)
  - Finite or infinite languages
  - Example:
    - {0, 00, 11, 000, 011, 101, 110, 0000, 0011, ...}
    - {int, double, char}
    - {a, b, a+a, a+b, b+a, b+b, ... a-(b+b), ..}

How we can describe a language?

- List of elements - applicable only to finite languages (most programming languages are not...)
- Description in *spoken* language
  - Vague, hard to really process by a computer.
  - Sometimes all we can get...
- Formal *generative* systems - **grammars**
  - Grammars compose from a set of rules, based on them, we are ultimately able to generate all language's words.
  - Often easily readable and understandable by a programmer.
- Formal *detection* systems - **automatons**
  - Automaton define a system, that when executed with some input word provides an answer (or computes indefinitely) to question: If the word belong to recognized language.
  - Often, easy to simulate the detection on a computer (frequently with a good run-time performance).

What we will do in several next lectures will be to chose a proper generative systems to describe the source language and then convert them to appropriate detection systems.

Generative Grammars

Definition

A **generative grammar** is a tuple $\mathcal{G} = (\Pi, \Sigma, S, P)$, where

- Π is a finite set of nonterminals
- Σ is a finite set of terminals, $\Pi \cap \Sigma = \emptyset$
- $S \in \Pi$ is the initial nonterminal
- P is a finite set of rules of the form $\alpha \rightarrow \beta$, where $\alpha \in (\Pi \cup \Sigma)^* \Pi (\Pi \cup \Sigma)^*$ and $\beta \in (\Pi \cup \Sigma)^*$.

Example of a rule:

$$\mathrm{CaECb} \rightarrow \mathrm{bDFbBDaC}$$

Remark: This type of grammar is also called **type-0** grammars, **unrestricted** grammars, or **phrase structure grammars**.

Generative Grammars

Let us assume that we have a generative grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$.
Relation $\Rightarrow \subseteq (\Pi \cup \Sigma)^* \times (\Pi \cup \Sigma)^*$:

- $\mu_1 \alpha \mu_2 \Rightarrow \mu_1 \beta \mu_2$ if $\alpha \rightarrow \beta$ is a rule from P

Example: If $(BcE \rightarrow DDaBb) \in P$ then

$$CaBCBcEAccABb \Rightarrow CaBCDDaBbAccABb$$

A **language** $\mathcal{L}(\mathcal{G})$ generated by a grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$ is the set of all words over alphabet Σ that can be derived by some derivation from the initial nonterminal S using rules from P, i.e.,

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

Context-sensitive Grammars

Context-sensitive grammars, also called **type-1** grammars, are a special case of generativních grammars.
A grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$ is called **context-sensitive** if all its rules (with one exception given below) are of the form

$$\alpha X \beta \rightarrow \alpha \gamma \beta$$

where $X \in \Pi$, $\alpha, \beta, \gamma \in (\Pi \cup \Sigma)^*$, with $|\gamma| \geq 1$.
The only exception is that the grammar can contain the rule $S \rightarrow \varepsilon$.
If $\mathcal{G}$ contains this rule then the initial nonterminal S can not occur on the right-hand side of any rule.
An example of a rule:

$$BaEC \rightarrow BaDAcBC$$

Context-free Grammars

Another special type of generative grammars are **context-free grammars**.
Context-free grammars are also called **type-2** grammars.
A grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$ is **context-free** if all its rules are of the form

$$X \to \gamma$$

where $X \in \Pi$, $\gamma \in (\Pi \cup \Sigma)^*$.
A example of a rule:

$$C \to DaBBc$$

Regular grammars

Definition

A grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$ is **right regular** if all rules in P are of the following forms (where $A, B \in \Pi$, $a \in \Sigma$): $A \to B$, $A \to aB$, $A \to \varepsilon$

Definition

A grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$ is **left regular** if all rules in P are of the following forms (where $A, B \in \Pi$, $a \in \Sigma$): $A \to B$, $A \to Ba$, $A \to \varepsilon$

Definition

A grammar $\mathcal{G}$ is **regular** if it right regular or left regular.

Regular grammars are also called **type-3** grammars.

Chomsky Hierarchy

So according to the types of rules that can be used in a grammar, the grammars can be divided into these four types:

- **Type-0** — General **generative grammars**
  no restrictions on the rules
- **Type-1** — **Context-sensitive grammars**
  rules of the form $\alpha X \beta \rightarrow \alpha \gamma \beta$, where $|\gamma| \geq 1$
  (An exception is possible rule $S \rightarrow \varepsilon$, but then S does not occur on the right-hand side of any rule.)
- **Type-2** — **context-free grammars**
  rules of the form $X \rightarrow \gamma$
- **Type-3** — **regular grammars**
  rules of the form $X \rightarrow wY$ (resp. $X \rightarrow Yw$) or $X \rightarrow w$

where $\alpha, \beta, \gamma \in (\Pi \cup \Sigma)^*$, $X \in \Pi$, and $w \in \Sigma^*$

Chomsky Hierarchy - summary

- **Type-0** — **recursively enumerable** languages:
  - unrestricted generative grammars
  - Turing machines (deterministic, nondeterministic)
  - We are *unable* to compute, if words belong to a language.
- **Type-1** — **context-sensitive** languages:
  - context-sensitive grammars
  - nondeterministic linear bounded automata
  - Containing real programming languages.
  - We are unable to analyze *effectively*, if words belong to a language.
- **Type-2** — **context-free** languages:
  - context-free grammars
  - nondeterministic pushdown automata
  - Effective analysis (especially for some sub-classes), if words belong to a language.
- **Type-3** — **regular** languages:
  - regular grammars
  - finite automata (deterministic, nondeterministic)
  - regular expressions
  - Even better performance

Chomsky Hierarchy

- An example of a language that is context-free but is not regular:

$$\{a^n b^n \mid n \geq 1\}$$

- An example of a language that is context-sensitive but is not context-free:

$$\{a^n b^n c^n \mid n \geq 1\}$$

Lexical analysis - basic definitions

- Lexical analysis (lexing or tokenization) is the process of converting an input (a text, sequence of symbols) into a sequence of lexical symbols (tokens).
- A program that performs lexical analysis may be termed a lexer, tokenizer, or scanner (sometimes it is also used for the first step in lexical analysis).
- Basic definitions:
  - Lexemes - it is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.
  - Token (lexical symbol) category (class, name) - identifier, number, operator,...
  - **Token** (lexical symbol) is a pair (token category, token value)
    - Usually, a token value is the corresponding lexeme - `(identifier, "start")`.
    - Sometime, there is no value - `(if, "")`
    - Sometime, the value is *processed* and the lexeme is converted to other value `(integer, 60)`

Lexical analysis I

- A set of rules, that defines the lexical syntax is usually called lexical grammar.
- The lexical syntax is **on purpose** a **regular language**.
  - Best tool to define lexemes are regular expressions.
  - Best tool to recognize lexemes in the input are finite automatons.
- Usually, *the lexical grammar* is a set of named definitions:

```
MUL : '*' ;
ADD : '+' ;
ID : [a-zA-Z]+ ;          // match identifiers
INT : [0-9]+ ;            // match integers
NEWLINE:'\r'? '\n' ;      // return newlines to parser
WS : [ \t]+ -> skip ;     // toss out whitespace
```

- These definitions often have a priority associated with them.

Lexical analysis II

- In theory, These named regular definitions are then converted to Deterministic Finite Automaton (DFA).
  - Easy is to convert regular expressions to Non-deterministic Finite Automaton (NFA).
  - We know an algorithm to convert NFA $\rightarrow$ DFA.
  - By finite states, we can distinguish the original definitions.
- When we want identify tokens, usually from left to right, *the scanner* repeatedly tries to consume **the longest sequence** that corresponds to some of regular definitions.
  - `intx` - is identifier not a keyword.
- When the lexeme is identified, we can form the resulting token.
  - The value in the token is then the text of the lexeme.
- Optionally, we can **compute** the value in the token.
  - Some tools allow to add *some code* to the definition.
  - This code is then executed by *Evaluator* to get the real value in token.

Lexical analysis - implementation I

- Before, we described something like an ideal case based on theory for lexical analysis → not really like it **must** be implemented.
- For simple lexical grammars, *direct implementation* is still good option.
- It maybe also the best option for some **special cases**, for example some special treatment of indentations.

```
SkipSpaces();
if (Char.IsDigit(ch))
{
  stringAttr="";
  while (Char.IsNumber(ch))
  {
    stringAttr+=ch;
    getch();
  }
  numberAttr=Int32.Parse(stringAttr);
  return Tokens.NUMBER;
}
if (Char.IsLetter(ch)) {
  stringAttr="";
  while (Char.IsLetterOrDigit(ch))
  {
    stringAttr+=ch;
    getch();
  }
  if(stringAttr.Equals("div"))
    return Tokens.DIV;
  return Tokens.IDENT;
}
```

Lexical analysis - implementation II

- We can implement some kind of previously described theory (DFK, regular expressions → DFK,..), but it is impractical.
- Solution: **Use some tool.**
  - Specialized lexical analyzer: Lex(Flex)
  - Lexical + Syntactic analyzer: JavaCC, ANTLR, Coco/R, YACC(Bison)

```
SKIP :
{
    " " | "\r" | "\t"
}
TOKEN :
{
    < ADD: "+" > | < SUB: "-" > |   < MUL: "*" > | < DIV: "/" > | < MOD: "mod" >
}
TOKEN :
{
    < CONSTANT: ( <DIGIT> )+ > |   < #DIGIT: ["0" - "9"] >
}
```

Syntactic analysis I

- For **syntactic analysis** (syntax analysis, parsing) will use context-free languages.
  - Best human readable way, how to define such language is **a context-free grammar** (CFG).
  - Best way how to determine, that an input belong to a context-free language is **a push-down automaton** (PDA).
- What we want from the syntactic analyzer?
  - Check if the input is correct.
  - Recognize defined syntactic constructs in the input.
  - Build the derivation tree (for given input and context-free grammar).
    - Or some other form used in the next compiler's steps.
- How to do that? Let's do some brainstorming, which tools we already have.
  1. We can convert CFG to PDA $\rightarrow$ use it to check if the input is correct.
  2. If the input belongs to the language generated by the grammar, there is a derivation that generates the input.
  3. We can search for it, but it will be **time consuming** $\rightarrow$ we need to do some sort of *backtracking*.

Syntactic analysis II

- While universal methods are possible, they are rarely used in practice due to being *slow*.
- There are sub-classes of context free-grammars for which we can do the analysis *faster*.
  - The main idea behind them is: We want to somehow *reduce the ambiguity*.
- Two main classes are:
  - Top-down parsing (LL grammars)
    - Tokens are consumed from left to right.
    - Based on finding the left-most derivations of an input-stream by searching for parse trees using a top-down expansion of the given formal grammar rules.
    - ANTLR, JavaCC, Coco/R
  - Bottom-up parsing (LR grammars)
    - Again, tokens are consumed from left to right, but we are using (usually) right-most derivation.
    - A parser can start with the input and attempt to rewrite it to the start symbol
    - There are several variants of LR parsers: SLR parsers, LALR parsers, Canonical LR(1) parsers, Minimal LR(1) parsers, GLR parsers, ...
    - GNU Bison (Yacc)

Syntactic analysis III

What are common constructs appearing in programming languages? How do we capture these constructs in a grammar?

- Sequence ($S \rightarrow A\ B$)
- Alternative ($S \rightarrow A \mid B$)
- Hierarchy ($E \rightarrow (\ E\ )$)
- Iteration ($AAA...$)

$$S \rightarrow SA \mid A$$

$$S \rightarrow AS \mid A$$

$$S \rightarrow AS'$$
$$S' \rightarrow AS' \mid \varepsilon$$

- Iteration with some delimiter ($A, A, A, ...$)

$$S \rightarrow S, A \mid A$$

$$S \rightarrow A, S \mid A$$

$$S \rightarrow AS'$$
$$S' \rightarrow , AS' \mid \varepsilon$$

LL(1) Grammars - Motivation I

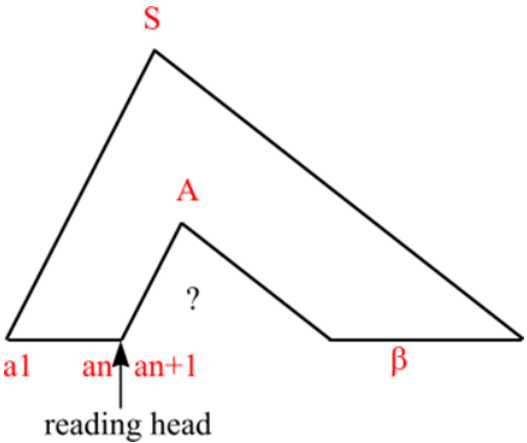
- In these lectures, we will focus on LL(1) grammars.
  - They are probably the simplest to understand and learn.
  - It is very easy to implement (from scratch) a parser based on LL(1) grammars using recursive descent.
- Informally, the difference between LL(1) and context-free grammars $\rightarrow$ removed ambiguity.
- First, they are using left-most derivation - whats that?

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow n \mid (E)$$

$$\underline{E} \Rightarrow \underline{E} + T \Rightarrow \underline{T} + T \Rightarrow \underline{F} + T \Rightarrow n + \underline{T} \Rightarrow$$
$$n + \underline{T} * F \Rightarrow n + \underline{F} * F \Rightarrow n + n * \underline{F} \Rightarrow n + n * n$$

- As a consequence, we know exactly, which non-terminal we **need** to rewrite.
- We do **not know** which of its right sides we should use.

LL(1) Grammars - Motivation II



- Input: $a_1, a_2, ..., a_n, a_{n+1}, ..., a_i$
- Depicted point in analysis: $S \Rightarrow^* a_1, a_2, ...a_n A \beta$
- What we need to decide at this point? $A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_j$
- We need to chose **a right side**.
- What information we can use?
  - Non-terminal A **needs** to generate a sequence starting with a_{n+1}.
- We have defined $LL(1)$ grammars in a way, that this information is enough.

LL(1) Grammars - Set FIRST

- Consider following examples:

$$A \to aB \mid bC \mid c$$

$$A \to Ba \mid c$$
$$B \to bB \mid d$$

- $\mathrm{FIRST}(\alpha)$ is defined as a set of terminal symbols which are the first letters of strings derived from α.
- We will chose corresponding right side based on its FIRST set.

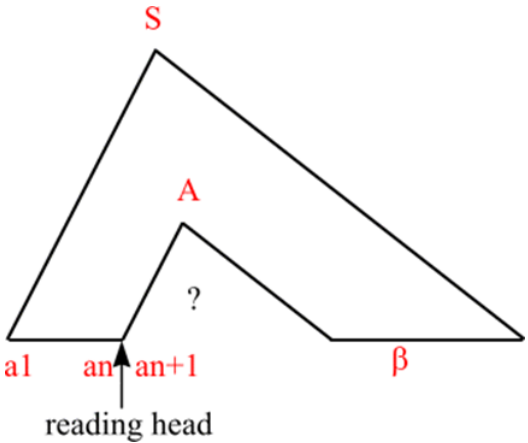
Definition

Lets have a context-free grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$ and $\alpha \in (\Pi \cup \Sigma)^*$ then
$\mathrm{FIRST}(\alpha) = \{a \in \Sigma \mid \alpha \Rightarrow^* a\beta, \beta \in (\Pi \cup \Sigma)^*\} \cup \{\varepsilon \mid \alpha \Rightarrow^* \varepsilon\}$

LL(1) Grammars - How to compute FIRST

- Before we start with an algorithm.
  - Lets have a context-free grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$ and $\alpha \in (\Pi \cup \Sigma)^*$
  - We will need a set of non-terminal that can be rewritten as an empty word (is nullable): $\Pi_\varepsilon = \{A \in \Pi \mid A \Rightarrow^* \varepsilon\}$
- We can compute $\mathrm{FIRST}(\alpha)$ as:
  1. If $\alpha = a\beta$ where $a \in \Sigma,\ \beta \in (\Pi \cup \Sigma)^*$ then $\mathrm{FIRST}(\alpha) = \{a\}$
  2. if $\alpha = \varepsilon$ then $\mathrm{FIRST}(\alpha) = \{\varepsilon\}$
  3. If $\alpha = X\beta$ where $X \in \Pi \setminus \Pi_\varepsilon,\ \beta \in (\Pi \cup \Sigma)^*,\ X \to \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$ then $\mathrm{FIRST}(\alpha) = \cup_{i \in \{1..n\}} \mathrm{FIRST}(\gamma_i) \setminus \{\varepsilon\}$
  4. If $\alpha = X\beta$ where $X \in \Pi_\varepsilon,\ \beta \in (\Pi \cup \Sigma)^*,\ X \to \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$ then $\mathrm{FIRST}(\alpha) = (\cup_{i \in \{1..n\}} \mathrm{FIRST}(\gamma_i) \setminus \{\varepsilon\}) \cup \mathrm{FIRST}(\beta)$

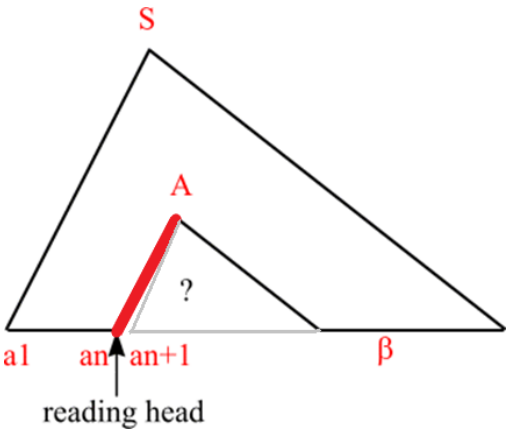
LL(1) Grammars - Using FIRST sets



- Lets return to our original problem.
- Input: $a_1, a_2, ..., a_n, a_{n+1}, ..., a_i$
- State: $S \Rightarrow^* a_1, a_2, ...a_n A \beta$
- We need to chose **a right side**.
  $A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$
- We will compute:
  $FIRST(\alpha_1), FIRST(\alpha_1), ...FIRST(\alpha_n)$
- If these sets are disjoint, we find a set where $a_{n+1} \in FIRST(\alpha_j)$, $j \leq n$, we will use rule: $A \rightarrow \alpha_j$
- Because this rule is **the only one**, that is able to generate a_{n+1} at the beginning.

LL(1) Grammars - Set FOLLOW I

Does it solve all our problems? What if A has a right side: $\alpha_i \Rightarrow^* \varepsilon$, when it will be used if we want to rewrite it to ε?



- Input: $a_1, a_2, ..., a_n, a_{n+1}, ..., a_i$
- State: $S \Rightarrow^* a_1, a_2, ...a_n A\beta$
- Where a_{n+1} can be found? $\rightarrow$ It **follows** right after $A \rightarrow$ It is in $\text{FIRST}(\beta)$
- We will compute new set FOLLOW, for each **non-terminal**.
- We will use it to determine, if the right side will be rewritten to ε.
- Notice: Just one right side $\alpha_i \Rightarrow^* \varepsilon$, also $\varepsilon \in \text{FIRST}(\alpha_i)$.

LL(1) Grammars - Set FOLLOW II

- Consider following example:

$$S \rightarrow bAc$$
$$A \rightarrow aA \mid \varepsilon$$

- $FOLLOW(A)$ is defined as a set of terminal symbols, that directly follows non-terminal A in any sentential form that can be reached from the starting non-terminal.

Definition

Lets have a context-free grammar $\mathcal{G} = (N, \Sigma, S, P)$ and non-terminal $A \in N$ then $FOLLOW(A) = \{a \in \Sigma \mid S \Rightarrow^* \alpha A \beta,\ \alpha, \beta \in (N \cup \Sigma)^*,\ a \in FIRST(\beta)\}$

- Often, we need to know that we have reached the end of the input.
- This is solved by special character($\$$), that will mark the end of the input.
- Then we need to modify our definition of FOLLOW, it will use $\mathcal{G}' = (N,\ \Sigma \cup \{\$\},\ S',\ P \cup \{S' \rightarrow S\$\})$ that comes from $\mathcal{G}$.

LL(1) Grammars - How to compute FOLLOW

- Before we start with an algorithm.
  - Lets have a context-free grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$ and $A \in \Pi^*$
  - We do not extend $\mathcal{G}$, as was mentioned before, but *the same result* can be achieved by adding \$ to $\mathrm{FOLLOW}(S)$ at the beginning of the computation.
- We can compute FOLLOW sets as:
  - Add to $\mathrm{FOLLOW}(S)$ the character marking the end of the input (\$), i.e. $\$ \in \mathrm{FOLLOW}(S)$.
  - To compute $\mathrm{FOLLOW}(A)$, for all rules: $X \to \alpha A \beta$, where $X \in \Pi,\ \alpha, \beta \in (\Pi \cup \Sigma)^*$ do:
    1. Add $\mathrm{FIRST}(\beta) \setminus \{\varepsilon\}$ into $\mathrm{FOLLOW}(A)$, i.e. $\mathrm{FIRST}(\beta) \setminus \{\varepsilon\} \subseteq \mathrm{FOLLOW}(A)$
    2. If $\beta \Rightarrow^* \varepsilon$ then add $\mathrm{FOLLOW}(X)$ to $\mathrm{FOLLOW}(A)$, i.e. $\mathrm{FOLLOW}(X) \subseteq \mathrm{FOLLOW}(A)$

- Note, that both algorithms (for FIRST and FOLLOW) are recursive. If we want to implement them, then we will probably use slightly different version.

LL(1) Grammars - Example I

$$A \to BCd \mid aB$$
$$B \to bB \mid \varepsilon$$
$$C \to cA \mid \varepsilon$$

$$A^{d,\$} \to BCd^{\mathbf{b,\ c,\ d}} \mid aB^{\mathbf{a}}$$
$$B^{c,d,\$} \to bB^{\mathbf{b}} \mid \varepsilon^{\boldsymbol{\varepsilon}}$$
$$C^{d} \to cA^{\mathbf{c}} \mid \varepsilon^{\boldsymbol{\varepsilon}}$$

- $\mathrm{FIRST}(BCd) = (\mathrm{FIRST}(bB) \cup \mathrm{FIRST}(\varepsilon)) \setminus \{\varepsilon\} \cup \mathrm{FIRST}(Cd) = \{b\} \cup \mathrm{FIRST}(Cb)$
  $= \{b\} \cup ((\mathrm{FIRST}(cA) \cup \mathrm{FIRST}(\varepsilon)) \setminus \{\varepsilon\} \cup \mathrm{FIRST}(d) = \{b\} \cup \{c\} \cup \mathrm{FIRST}(d)$
  $= \{b,\ c\} \cup \{d\} = \{b,\ c,\ d\}$
- $\mathrm{FOLLOW}(A) = \mathrm{FIRST}(\varepsilon) \setminus \{\varepsilon\} \cup \mathrm{FOLLOW}(C)$ (from rule: $C \to cA$) $= \mathrm{FOLLOW}(C)$
  $= \mathrm{FIRST}(d) \setminus \{\varepsilon\}$ (from rule: $A \to BCd$) $= \{d\}$
  - $\{d,\ \$\}$ - If we consider A as starting non-terminal.

LL(1) Grammars - Example II

$$A \to BCd \mid aB$$
$$B \to bB \mid \varepsilon$$
$$C \to cA \mid \varepsilon$$

$$A^{d,\$} \to BCd^{\mathbf{b,\ c,\ d}} \mid aB^{\mathbf{a}}$$
$$B^{c,d,\$} \to bB^{\mathbf{b}} \mid \varepsilon^{\boldsymbol{\varepsilon}}$$
$$C^{d} \to cA^{\mathbf{c}} \mid \varepsilon^{\boldsymbol{\varepsilon}}$$

FIRST for non-terminals

	A	B	C	a	b	c	d
A		*	*	*			*
B					*		
C						*	

	A	B	C	a	b	c	d
A		*	*	*	*	*	*
B					*		
C						*	

FOLLOW for non-terminals

	A	B	C	a	b	c	d	$
A			*					*
B	*	*				*	*	
C							*	

	A	B	C	a	b	c	d	$
A			*				*	*
B	*	*				*	*	*
C							*	

LL(1) Grammars - Definition I

- We started with an idea, that we are reading the input from left to right, we are building left-most derivation and **the problem** is, we want to decide, which right side of the left-most non-terminal we should use.
- This decision primarily based on FIRST, for nullable rules we will use also FOLLOW.
- Formally, we can define SELECT sets, that will be used for this decision.

Definition

Lets have a context-free grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$ and a rule $(A \rightarrow \alpha) \in P$ where $A \in \Pi,\ \alpha \in (\Pi \cup \Sigma)^*$, then:

- if $\alpha \Rightarrow^* \varepsilon$ then $\textbf{SELECT}(A \rightarrow \alpha) = (\mathrm{FIRST}(\alpha) \setminus \{\varepsilon\}) \cup \mathrm{FOLLOW}(A)$
- else $\textbf{SELECT}(A \rightarrow \alpha) = \mathrm{FIRST}(\alpha)$

LL(1) Grammars - Definition II

- LL(1)
  - The first L stands for scanning the input from left to right.
  - The second L stands for producing a leftmost derivation.
  - 1 stands for using one input symbol of look-ahead at each step to make parsing action decision.
- Informally, LL(1) grammars are context-free grammars where the decision which right-side we should use (in the left-most derivation) is **deterministic** $\rightarrow$ it is not ambiguous and not left-recursive.

Definition

A context-free grammar $\mathcal{G} = (N, \Sigma, S, P)$ is **LL(1)** if and only if for every $A \in N$ and every $\alpha, \beta \in (N \cup \Sigma)^*$, such that $(A \rightarrow \alpha) \in P$ and $(A \rightarrow \beta) \in P$ and $\alpha \neq \beta$ we have:

- $\mathrm{FIRST}(\alpha) \cap \mathrm{FIRST}(\beta) = \emptyset$;
- if $\alpha \Rightarrow^* \varepsilon$ then $\mathrm{FIRST}(\beta) \cap \{\varepsilon\} = \emptyset$ and $\mathrm{FIRST}(\beta) \cap \mathrm{FOLLOW}(A) = \emptyset$.

LL(1) Grammars - Definition III

- Alternatively, we can define LL(1) grammars using SELECT sets.

Definition

A context-free grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$ is **LL(1)** if and only if for every $A \in \Pi$ and every $\alpha, \beta \in (\Pi \cup \Sigma)^*$, such that $(A \to \alpha) \in P$ and $(A \to \beta) \in P$ and $\alpha \neq \beta$ we have $\mathrm{SELECT}(A \to \alpha) \cap \mathrm{SELECT}(A \to \beta) = \emptyset$.

$$
\begin{aligned}
A^{\,d,\$} &\to BCd^{\,b,\,c,\,d} \mid aB^{\,a} \\
B^{\,c,d,\$} &\to bB^{\,b} \mid \varepsilon^{\,\varepsilon} \\
C^{\,d} &\to cA^{\,c} \mid \varepsilon^{\,\varepsilon}
\end{aligned}
\qquad\qquad
\begin{aligned}
S^{\,*,),\$} &\to S * E^{\,(,\,i} \mid E^{\,(,\,i} \\
E^{\,*,),\$} &\to TR^{\,(,\,i} \\
R^{\,*,),\$} &\to +TR^{\,+} \mid \varepsilon^{\,\varepsilon} \\
T^{\,+,?,*,),\$} &\to (S)^{\,(} \mid T?^{\,(,\,i} \mid i^{\,i}
\end{aligned}
$$

Try to solve, if the grammars are LL(1).

LL(1) Grammars - Definition III

- Alternatively, we can define LL(1) grammars using SELECT sets.

Definition

A context-free grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$ is **LL(1)** if and only if for every $A \in \Pi$ and every $\alpha, \beta \in (\Pi \cup \Sigma)^*$, such that $(A \to \alpha) \in P$ and $(A \to \beta) \in P$ and $\alpha \neq \beta$ we have $\mathrm{SELECT}(A \to \alpha) \cap \mathrm{SELECT}(A \to \beta) = \emptyset$.

$$
\begin{aligned}
A^{\,d,\$} &\to BCd^{\,b,\,c,\,d} \mid aB^{\,a} \\
B^{\,c,d,\$} &\to bB^{\,b} \mid \varepsilon^{\,\varepsilon} \\
C^{\,d} &\to cA^{\,c} \mid \varepsilon^{\,\varepsilon}
\end{aligned}
$$

It is LL(1) Grammar

$$
\begin{aligned}
S^{\,*,),\$} &\to S * E^{\,(,\,i} \mid E^{\,(,\,i} \\
E^{\,*,),\$} &\to TR^{\,(,\,i} \\
R^{\,*,),\$} &\to +TR^{\,+} \mid \varepsilon^{\,\varepsilon} \\
T^{\,+,?,*,),\$} &\to (S)^{\,(} \mid T?^{\,(,\,i} \mid i^{\,i}
\end{aligned}
$$

It is NOT LL(1) Grammar

Notice, what are the problems that are the cause that the grammar is not LL(1).

LL(1) Grammars - Parser I

Definition

A **pushdown automaton** (**PDA**) is a tuple $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ where

- Q is a finite non-empty set of states
- Σ is a finite non-empty set called an input alphabet
- Γ is a finite non-empty set called a stack alphabet
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ is a (nondeterministic) transition function
- $q_0 \in Q$ is the initial state
- $Z_0 \in \Gamma$ is the initial stack symbol

- A **configuration** of a PDA is a triple: (q, w, α) where $q \in Q$, $w \in \Sigma^*$, and $\alpha \in \Gamma^*$.
- An **initial configuration** is a configuration (q_0, w, Z_0), where $w \in \Sigma^*$.
- Language recognised by non-deterministic PDA are exactly context-free languages.
- Deterministic PDA are less powerful non-deterministic one.

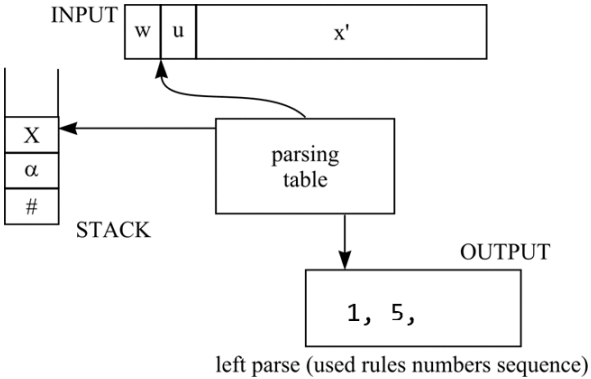
LL(1) Grammars - Parser II

The question is: How to map LL(1) grammar on PDA? How does the algorithm transforming context-free grammar to PDA works?

- We are creating a single state PDA where its transition relation is constructed as follows:
  1. *expand* - $(1, \varepsilon, A, 1, \alpha)$ for each rule: $A \rightarrow \alpha$
  2. *match* - $(1, a, a, 1, \varepsilon)$ for each symbol: $a \in \Sigma$
- If the PDA accepts by empty stack, its initial stack symbol is the grammar's start symbol.
- At each *step*, the constructed NPDA can *see*:
  - the current input symbol;
  - the topmost stack symbol.
- The whole idea behind LL(1) grammars was to be able to do this decision *deterministically* $\rightarrow$ interesting observation is that constructed automaton follows the same *idea*.

LL(1) Grammars - Parser III

- We will do a little *trick* and define a slightly more *advanced* automaton $\rightarrow$ it will have output $\rightarrow$ it will be leftmost derivation
  - We can create *real* PDA, if we really need to...



LL(1) Grammars - Parser IV

- We have LL(1) grammar $\mathcal{G} = (N, \Sigma, S, P)$
  - Assume, that all rules $r \in P$ are numbered, i.e. there is a function $p : P \to \mathbb{N}$ that assigns a unique number to each rule.
- Following previous information, we want PDA where:
  - There is just one state **(we can omit it)**.
  - The input alphabet is $\Sigma \cup \{\$\}$, where $\$$ marks the end of input.
  - The stack alphabet is $\Sigma \cup N \cup \{\#\}$, we will use PDA that **accepts by** reaching $\#$ on stack.
  - The initial configuration is $(w\$,\ S\#,\ \varepsilon)$ *//(input, stack, output)*
- Parsing table $M : (\Sigma \cup N \cup \{\#\}) \times (\Sigma \cup \{\$\}) \to \{\mathtt{expand\ i}, \mathtt{pop}, \mathtt{accept}, \mathtt{error}\}$ where:
  - **expand i**: $(w,\ A\beta,\ \pi) \vdash (w,\ \alpha\beta,\ \pi\ i)$, for $i = p(A \to \alpha)$
  - **pop**: $(aw,\ a\beta,\ \pi) \vdash (w,\ \beta,\ \pi)$
  - **error** - syntactic error
  - **accept** - computation is finished, output contains left-most derivation

LL(1) Grammars - Parser V

Building the parsing table $M : (\Sigma \cup \Pi \cup \{\#\}) \times (\Sigma \cup \{\$\})$

- For all rules $(A \rightarrow \alpha) \in P$ where $i = p(A \rightarrow \alpha)$ and for all $x \in SELECT(A \rightarrow \alpha)$ is $M[A, x] = expand\ i$.
- For all $a \in \Sigma$ is $M[a, a] = pop$
- $M[\#, \$] = accept$
- All unassigned fields in M are set to $error$.

LL(1) Grammars - Parser - Example

$$A \overset{d,\$}{\rightarrow} BCd \overset{b,\ c,\ d}{_1} \mid aB \overset{a}{_2}$$
$$B \overset{c,d,\$}{\rightarrow} bB \overset{b}{_3} \mid \varepsilon \overset{\varepsilon}{_4}$$
$$C \overset{d}{\rightarrow} cA \overset{c}{_5} \mid \varepsilon \overset{\varepsilon}{_6}$$

	a	b	c	d	$
A	e2	e1	e1	e1	
B		e3	e4	e4	e4
C			e5	e6	
a	pop				
b		pop			
c			pop		
d				pop	
#					acc

$(bcad\$,\ A\#,\ \varepsilon) \overset{e1}{\vdash} (bcad\$,\ BCd\#,\ 1) \overset{e3}{\vdash} (bcad\$,\ bBCd\#,\ 1\ 3) \overset{pop}{\vdash} (cad\$,\ BCd\#,\ 1\ 3)$
$\overset{e4}{\vdash} (cad\$,\ Cd\#,\ 1\ 3\ 4) \overset{e5}{\vdash} (cad\$,\ cAd\#,\ 1\ 3\ 4\ 5) \overset{pop}{\vdash} (ad\$,\ Ad\#,\ 1\ 3\ 4\ 5) \overset{e2}{\vdash}$
$(ad\$,\ aBd\#,\ 1\ 3\ 4\ 5\ 2) \overset{pop}{\vdash} (d\$,\ Bd\#,\ 1\ 3\ 4\ 5\ 2) \overset{e4}{\vdash} (d\$,\ d\#,\ 1\ 3\ 4\ 5\ 2\ 4) \overset{pop}{\vdash}$
$(\$,\ \#,\ 1\ 3\ 4\ 5\ 2\ 4) \overset{acc}{\vdash}$ **(1 3 4 5 2 4)**

How to Get LL(1) Grammar?

- Until this point, *everything* that was defined (for LL(1) - FIRST, FOLLOW, parsing table,...) can be *computed* (there is an algorithm).
- Embarrassingly, there is no algorithm that converts **any** context-free grammar to LL(1).
  - For some context-free languages there is no LL(1) grammar generating them.
  - Some (especially ambiguous) grammars can not be *transformed* to LL(1).
- What can be done?
  - Removing left recursion

    From: $A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$

    To: $A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$

    $A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$
  - Left factorization (removing common prefix)

    From: $A \to \beta\alpha_1 \mid \beta\alpha_2 \mid \ldots \mid \beta\alpha_n$

    To: $A \to \beta A'$

    $A' \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
  - Eliminating rules

How to build a Top-down parser

- Frequently used approach, how to implement top-down parser is **recursive descent**.
- A set of mutually recursive procedures where each such procedure implements one of the non-terminals of the grammar.
- It can be implemented using:
  - backtracking - it determines which production to use by trying each production in turn, possible for all grammars, but it terminates only for LL(k) grammars, may require exponential time.
  - **predictive parse** (form of non backtracking parser) - possible only for LL(k), linear time.
- LL(k) grammars - removes ambiguity - remove left recursion, left factored grammars.
- LL(1) grammars - we have used a *parsing table*, no recursion.

Recursive Descent for LL(1) Grammars I

- Lets have have LL(1) grammar $\mathcal{G} = (\Pi, \Sigma, S, P)$
- Terminals are represented by a function `expect`, that matches the current token against a predicted token.

```
Token token;  // current token

void expect(Token expectedToken)
{
    if( token == expectedToken ) token = scanner.NextToken();
    else Error();
}
```

Recursive Descent for LL(1) Grammars II

- Each non-terminal is represented by a function, that performs its analysis.
- Assume we have a non-terminal A with just one rule: $A \rightarrow x_1\ x_2\ \ldots\ x_n$ it will be represented by:

```
void A()
{
    // analysis of x1
    // analysis of x2
    ...
    // analysis of x3
}
```

The analysis of:

- $x \in \Sigma$ - calling function `expect`
- $x \in \Pi$ - calling function `x`.

Example: Lets have a non-terminal with just one rule: $E \rightarrow +\ T\ E\ ;$

```
void E()
{
    expect(Token.Plus);
    T();
    E();
    expect(Token.Semicolon);
}
```

Recursive Descent for LL(1) Grammars III

- What if the non-terminal have several right sides?
- Assume: $A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$
- For the decision we will use (again): $\mathrm{SELECT}(A, \alpha_i)$

```
void A() {
    if (Select("A", "alpha 1").Contains(token)) {
        // analysis of 1st symbol of alpha 1
        // analysis of 2nd symbol of alpha 1
        ...
    }
    if (Select("A", "alpha 2").Contains(token)) {
        // analysis of 1st symbol of alpha 2
        // analysis of 2nd symbol of alpha 2
        ...
    }
    ...
}
```

Recursive Descent for LL(1) Grammars - Example

$$E \rightarrow TE_1 \quad {}^{(,n}$$
$$E_1 \rightarrow +TE_1 \mid \varepsilon \quad {}^{+} \quad {}^{\$,)}$$
$$T \rightarrow FT_1 \quad {}^{(,n}$$
$$T_1 \rightarrow *FT_1 \mid \varepsilon \quad {}^{*} \quad {}^{\$,),+}$$
$$F \rightarrow (E) \mid n \quad {}^{(} \quad {}^{n}$$

```
void E() {
    T();
    E1();
}
```

```
void E1() {
    if (token == Token.Plus) {
        expect(Token.Plus);
        T();
        E1();
    } else if (token == Token.EOF
            || token == Token.RPar) {
        // epsilon
    } else {
        Error();
    }
}
...
```

Non-recursive Predicative Analysis for LL(1) I

- What if we want to implement for our $\mathrm{LL}(1)$ grammar mentioned PDA? $\rightarrow$ Not that hard.

- Syntactic analyzer will be *driven* by a parsing table. Main *memory* will be a stack.
- What expect to have:
  - Parsing table: $\mathrm{M}[A,\ \alpha]$
  - Stack: $\mathrm{push}(\alpha),\ \mathrm{pop}(),\ \mathrm{top}()$
  - Output: $\mathrm{output}(\mathrm{i})$
  - Controlling errors: $\mathrm{error}()$

Non-recursive Predicative Analysis for LL(1) II

Algorithm: Top-level of the predicative parser

```
push(#S);
a := scanner.NextSymbol();
do
    X := top() ;
    if X ∈ (Σ ∪ {$}) then
        if X = a then
            pop();
            a := scanner.NextSymbol();
        else error() ;
    else if M[X, a] = expandp_i then
        pop() ; // There is a rule numbered:  p_i :  X → Y_1 Y_2 ... Y_n
        push(Y_n Y_{n-1} ... Y_1) ;
        output(i);
    else error() ;
while X ≠ #;
```

Compiler-Compiler - Parser Generator

- Why to bother with implementation from scratch? → Use a tool: compiler-compiler (compiler-generator, parser generator)
- Various tools, based on various theoretical models (LALR(1), LL(k), ...).
- We are not *limited* by LL(1).
- The input is usually *a sort of* Extended BNF → the output is a program in programming language, in which you are implementing a compiler.
- There are plenty of such tools (https://en.wikipedia.org/wiki/Comparison_of_parser_generators)
  - ANTLR4 - AdaptiveLL(∗), *input*: EBNF *output*: C#, Java, Python, JavaScript, C++, Swift, Go, PHP *runs on*: JVM.
  - JavaCC - LL(k), *input*: EBNF *output*: Java, C++, JavaScript runs on: JVM.
  - Coco/R - LL(1), *input*: EBNF *output*: C, C++, C#, F#, Java, Ada, Object Pascal, Delphi, Modula-2, Oberon, Ruby, Swift, Unicon, Visual Basic .NET *runs on*: JVM, .NET.
  - GNU Bison - LALR(1), LR(1), IELR(1), GLR *input*: Yacc output: C, C++, Java .NET *runs on*: (GNU) All.

Compiler-Compiler - ANTLR4

We will use it for our project.

```
grammar Calculator;
// parser
start : expr | <EOF> ;

expr : '-' expr      # UMINUS
   | expr mulop expr # MULOPGRP
   | expr addop expr # ADDOPGRP
   | '(' expr ')'    # PARENGRP
   | NUMBER       # DOUBLE
   ;

addop : '+' | '-' ;
mulop : '*' | '/' | '%' ;

// lexer
NUMBER : ('0' .. '9') + ('.' ('0' .. '9') +)? ;
WS : [ \r\n\t] + -> skip ;
```

Compiler-Compiler - JavaCC

```
options {
   IGNORE_CASE = true; DEBUG_PARSER = true;
}
PARSER_BEGIN(Calc)
public class Calc {
  public static void main(String args[])
       throws ParseException
  { Calc parser = new Calc(System.in);
    parser.expr();
  }
}
PARSER_END(Calc)

SKIP :
{ " " | "\r" | "\t" }
TOKEN :
{ < EOL: "\n" > | < SEMICOLON: ";" > |
   <ADD: "+"> | <SUB: "-"> |
   <MUL: "*"> | <DIV: "/"> | <MOD: "mod"> }
TOKEN :
{  <CONSTANT: ( <DIGIT> )+ >
 | <#DIGIT: ["0" - "9"]> }

void expr() : { }
{ term() (( "+" | "-" ) term())* }

void term() : { }
{ factor() (("*" | "/" | "mod") factor())* }

void factor() : { }
{ <CONSTANT>  | "(" expr() ")" }
```

Compiler-Compiler - Bison (Yacc)

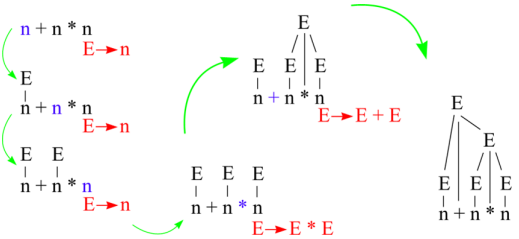
Just syntactic analyzer (Lex-Flex).

```
%{
#include <stdio.h>
#define YYSTYPE double
%}
%token NUMBER
%token PLUS MINUS TIMES DIVIDE POWER
%token LEFT RIGHT
%token END
%left PLUS
%left TIMES
%left NEG
%start Input
%%
Input: /* empty */ | Input Line;
Line: END
    | Expression END { printf("%f\n", $1);}
;
Expression:
     NUMBER { $$=$1; }
| Expression PLUS Expression { $$=$1+$3; }
| Expression TIMES Expression { $$=$1*$3; }
| MINUS Expression %prec NEG { $$=-$2; }
| LEFT Expression RIGHT { $$=$2; }
;
%%
int yyerror(char *s) {
  printf("%s\n", s);
}
int main() {
  yyparse();
}
```

LR Grammars I

- LR Grammars
  - Reading input from **L**eft to right.
  - **R**ight-most derivation in *reverse*.
- Bottom-up parser.
- Any LR(k) can be transformed to LR(1).
  - In practice we use LR(1).
  - Really we are using: SLR, LALR, GLR, Canonicla LR(1),...

- Larger class than LL(k).



LR Grammars II

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to n \mid (E)$$

$$\underline{E} \Rightarrow E + \underline{T} \Rightarrow E + T * \underline{F} \Rightarrow E + \underline{T} * n \Rightarrow$$
$$E + \underline{F} * n \Rightarrow \underline{E} + n * n \Rightarrow \underline{T} + n * n \Rightarrow$$
$$\underline{F} + n * n \Rightarrow n + n * n$$

Stack	Input	Output
	n + n * n $	
n	+ n * n $	F → n
F	+ n * n $	T → F
E	+ n * n $	E → T
E +	n * n $	
E + n	n * n $	F → n
E + F	* n $	T → F
E + T	* n $	
E + T *	n $	
E + T * n	$	F → n
E + T * F	$	T → T * F
E + T	$	E → E + T
E	$	accept

Table: Example of an LR analysis

Error Handling I

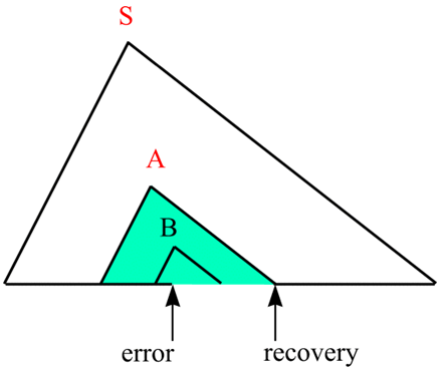
- What is an error? Where does it occurs? What are the examples of errors?
  - *Just Errors* - unexpected behaviour of the program $\rightarrow$ beyond the scope of this lecture.
  - Run-time Errors
  - Compile-time Errors $\leftarrow$ we will focuse on this.
- Compile-time Errors
  - Lexical analysis (not closed string, illegal character) - usually easy, small number of well defined errors.
  - *Syntactic analysis* (missing semicolon or parenthesis) - if we are talking about approaches to error handling, usually we are referring to handling syntax errors.
  - Semantic analysis (various type related errors) - more complex, implementation depends on other factors like used intermediate representation.
  - Logical errors (infinite loop, unreachable code)
- *Good error handling is not easy to achieve.*

Error Handling II

- Functions of Error Handler:
  - Error Detection
  - Error Report
  - Error Recovery
- Approaches to Syntax Errors Recovery:
  - **Panic mode** (the easiest and the most frequent way)
  - Phase level recovery (local corrections) - when an error is discovered, the parser performs local correction on the remaining input, so that the parser can continue.
  - Error productions - common errors are included as rules in the parser's grammar and solved.
  - Global corrections - the parser analyzes the whole input and tries to find *the closest* error-free version.
- History
  - In the past, the compilation was slow $\rightarrow$ error recovery was important $\rightarrow$ we want to find as many errors as possible.
  - Now, compilations time is usually not an issue $\rightarrow$ it is not that important $\rightarrow$ less but more precise reported errors is enough.

Panic Mode Error Recovery I

- What to do, if we encounter syntactic error?
  - We can stop the parser and report this error **(valid option)**.
  - We can try to **recover**.
- How can we recover after an error in LL(1) parser?



1. We need to define some *synchronizing tokens*, that *defines*, where we are in the grammar.
2. When we encounter an error, we will skip the input until the next synchronizing token.
3. we will search for the corresponding place in the current grammar's evaluation (derivation tree).
4. We will continue the analysis.

Panic Mode Error Recovery II

- It looks great, but What are the issues with the error recovery?
  - What should be the synchronizing token?
  - Are these synchronizing tokens fixed? Or they are changing trough the computation?
  - Conflicting requirements - the more ambitious error recovery, the less precise it is.
- The method, how we handle the problem of synchronizing tokens, is usually called *error recovery schema*.
  - For example, for LL(1) grammars we can use FIRST and FOLLOW sets to define synchronizing tokens.

Thank you for your attention

Marek Běhálek

VSB – Technical University of Ostrava

marek.behalek@vsb.cz

March 2, 2022

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE