Syntax and Semantic Analysis

Sudakshina Dutta

IIT Goa

29^{th} March, 2022

Tutorial-4

Show that the following grammar is $CLR(1)$, but not $LALR(1)$, $LL(1)$, $SLR(1)$

- $S \rightarrow Aa|bAc|Bc|bBa$
- $A \rightarrow d$
- $B \rightarrow d$

Note that while making table in CLR(1) / SLR(1) / LALR (1):

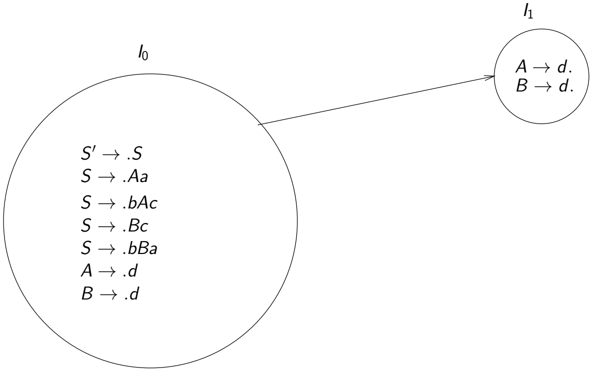
- Each state will definitely have at least 1 entry.

- Those states which currently have . at the end will have REDUCE moves.

$LL(1)$

- Both the productions $S \rightarrow Aa$ and $S \rightarrow Bc$ will go in the location $M[S, d]$ causing the grammar to be non-LL(1)

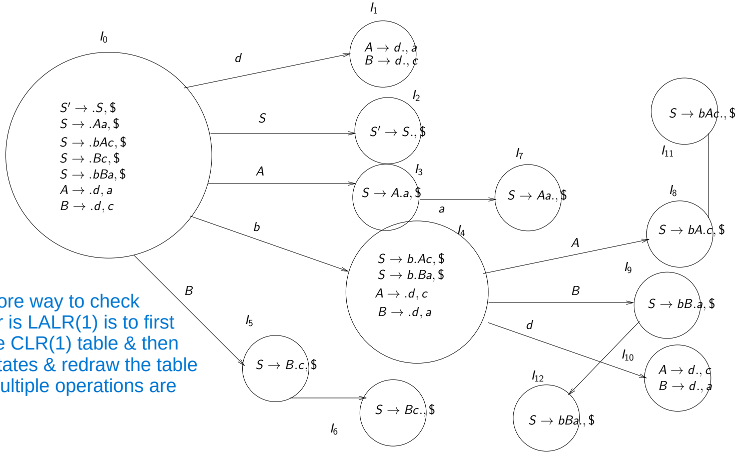
$SLR(1)$



Both the [1, a] and [1, c] have both the reduction moves $A \to d$ and $B \to d$

$CLR(1)$

Start by keeping $ as follow-up for the augmented production. For all the productions in the current state, Put follow-up based on the PARENT production that caused this to come. Specifically, follow-up of the Non terminal in that production. Also, While drawing transitions, The parent one's follow-up is not changed. But those productions which came becoz of the current parent might have their follow-ups changed.



Also, one more way to check if a grammar is LALR(1) is to first construct the CLR(1) table & then merge the states & redraw the table and see if multiple operations are feasible.

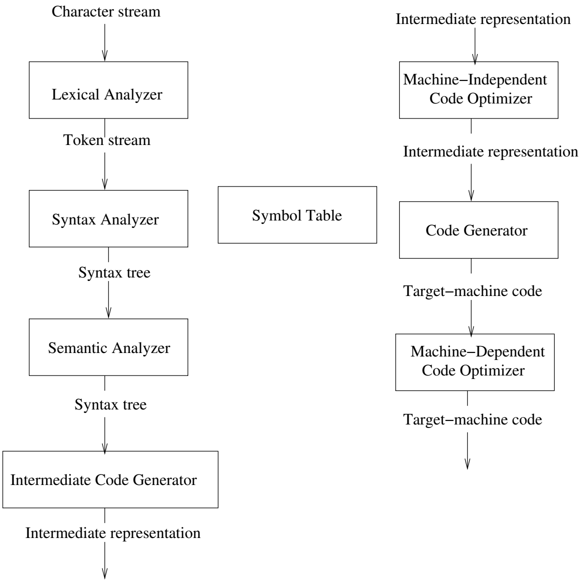
Both the [1, a] and [1, c] have both the reduction moves $A \to d$ and $B \to d$. Note that 1 is the merged state of I_1 and I_{10}

So, It is CLR(1) but NOT LALR(1).

Further analysis steps after parser

- An input program that is grammatically correct may still contain serious errors that would prevent compilation.
- To detect such errors, a compiler performs a further level of checking that involves considering each statement in its actual context.
- It can either performed alongside parsing or in a post- pass that traverses the IR produced by the parser.
- We call this analysis either "context-sensitive analysis," to differentiate it from parsing

The Phases of a Compiler



What is in a context ?

Suppose I want to use a name x. I need to know the following.

- What is stored in x ?
  - int, float, bool, etc or instances of compound types
- What is the size of x ?
  - single/double precision integer, for arrays number of dimension and size
- What are the types and number of arguments if x is a procedure ? What is the return type ?
- How long is the life time of the variable x ?

If there is no declaration, compiler derives all the above

Moving further beyond context-freeness

- The context-free grammar deals with syntactic categories rather than specific words
- The parser can recognize that the grammar allows a variable name to occur, and it can tell that one has occurred.
- However, the grammar has no way to match one instance of a variable name with another; that would require the grammar to specify a much deeper level of analysis
  - Conformance between declaration and use cannot be checked

Semantics of a programming language

- **Static semantics** : It includes those semantic rules that can be checked at compile time
- **Dynamic semantics** : The dynamic semantics (also known as execution semantics) of a language defines how and when the various constructs of a language should produce a program behavior

- Static semantics of programming language are checked by semantic analyzer
  - Variables are declared before use
  - Types and numbers of parameters match in declaration and use
  - Types match on both sides of assignments
- Compilers can generate codes to check dynamic semantics of the programming language
  - While processing arithmetic expressions, whether overflow occurs
  - Array limits are crossed
  - Recursion exceeds the stack

Why not Context Sensitive Grammar ?

- It might seem natural to consider the use of context-sensitive languages to perform context-sensitive checks

  — We have regular grammar to perform lexical analysis and context-free grammar to do parsing

- The productions of context-sensitive grammars are of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$, where $A \in NT$, $\alpha, \beta \in \{NT \cup T\}^*$ and $\gamma \in \{NT \cup T\}^+$

Why not Context Sensitive Grammar ?

- Context-sensitive grammars are not the right answer for two distinct reasons
  1. First, the problem of parsing a context-sensitive grammar is P-Space complete
  2. Second, many of the important questions are difficult, if not impossible, to encode in a context-sensitive grammar e.g., declaration before use

- One formalism for performing context-sensitive analysis is the attribute grammar, or attributed context-free grammar

  — Context-free grammar augmented by a set of rules where an attribute is computed in terms of values of other attributes

- If X is a symbol and a is one of its attributes, then $X.a$ is used denote it
- Can be of any kind: numbers, table references, or strings, etc.
- Example - production $E \rightarrow E_1 + T$ corresponds to semantic rule $E.code = E_1.code \mid\mid T.code \mid\mid ` + `$, say

An introduction to Type System

- A collection of properties with each data value. This is called a collection of properties — the value's type
- The range can be understood e.g., $-2^{31} \leq i \leq 2^{31}$
- Some types are defined and others are constructed
- Purpose : to specify the behavior very precisely

An introduction to type system

Production Rules :

$prog$: $funcDef$;
$funcDef$: $type$ ID '(' $argList$ ')' '{' $declList$ $stmtList$ '}'
$argList$: arg ',' arg | ϵ;
arg : $type$ ID;
$type$: INT | $FLOAT$;
$declList$: $declList$ $decl$ | $decl$;
$decl$: $type$ $varList$ $SEMICOLON$;
$varList$: ID $COMMA$ $varList$ | ID;
$stmtList$: $stmtList$ $stmt$ | $stmt$;
$stmt$: $assignStmt$ | $ifStmt$ | $whileStmt$;
$assignStmt$: ID '=' EXP $SEMICOLON$;
EXP : EXP '+' $TERM$ | EXP '-' $TERM$ | $TERM$;
$TERM$: $TERM$ '*' $FACTOR$ | $TERM$ '/' $FACTOR$ | $FACTOR$;
$FACTOR$: ID;
$ifStmt$: IF'('$bExp$')''{'$stmtList$ '}';
$bExp$: EXP $RELOP$ EXP;
$whileStmt$: $WHILE$'('$bExp$')''{'$stmtList$'}';

An introduction to type system

int a;

a	int

ID.name = a

ID.type = int

The purpose of type system

- To ensure run-time safety. The type system needs to be designed in a way so that subtle errors are caught
- Type safety is the extent to which a programming language discourages or prevents type errors
- Compiler derives types of expressions
- Sometimes compiler converts the types and sometimes some conversions are not permissible
  - $a + b$ is legal if both are integers. It is illegal if one is a record

The purpose of type system

- Prevention of illegal operations. For example, we can identify an expression 3 / "Compilers" as invalid, because the rules of arithmetic do not specify how to divide an integer by a string
- **Wild pointers** can arise when a pointer to one type object is treated as a pointer to another type
- **Buffer overflow** Out-of bound writes can corrupt the contents of objects already present on the heap

Strongly and weakly typed languages

Safety is a strong reason for using typed languages

- A language in which every expression can be assigned an unambiguous type is called a strongly typed language.
  - Java, Python, etc
- Weakly typed languages are those which allow type conversions and can lead to error
  - C, C++, etc
- There are also static and dynamic typed languages

Why static checking is preferred ?

- Runtime type checking imposes a large overhead
- It needs clear representation of types for each variable; each variable has both a value field and a tag field

```
// partial code for "a+b ⇒ c"
if (tag(a) = integer) then
     if (tag(b) = integer) then
          value(c) = value(a) + value(b);
          tag(c) = integer;
     else if (tag(b) = real) then
          temp = ConvertToReal(a);
          value(c) = temp + value(b);
          tag(c) = real;
     else if (tag(b) = ... ) then
          // handle all other types ...
     else
          signal runtime type fault
else if (tag(a) = real) then
     if (tag(b) = integer) then
          temp = ConvertToReal(b);
          value(c) = value(a) + temp;
          tag(c) = real;
     else if (tag(b) = real) then
          value(c) = value(a) + value(b);
          tag(c) = real;
```