Compilers

Spring term

Alfonso Ortega: alfonso.ortega@uam.es
Enrique Alfonseca: enrique.alfonseca@uam.es

Chapter 4: Syntactic analysis

4.0 Introduction
4.1 Bottom-up Analysis

Syntax Analyser

Concepts

- It analyses the context-independent structures in the programming language.
- A context-independent grammar is a quadruple:

$$< \Sigma_T, \Sigma_N, \text{axiom}, \text{rules} >$$

  - where
    - $\text{axiom} \in \Sigma_N$
    - $\text{rules} \in \wp(\Sigma_N \times \Sigma^*), \quad \Sigma = \Sigma_T \cup \Sigma_N$
  - In other words, the production rules have the following syntax:

$$A \rightarrow v \ , \ A \in \Sigma_N \wedge v \in (\Sigma_T \cup \Sigma_N)^*$$

- The syntax analyser is also called ***parser***.
- It usually controls all the actions performed by the compiler.
- The analysis can be understood as the procedure to find the "syntactic derivation tree" of the source program (if this is correct), which is a way of reducing the whole program to the grammar's axiom.
- Remember that, after the morphological analysis, the terminal symbols in the grammar are the syntactic units.

3

Language features which are not context-independent

Declaring identifiers before its use *[Aho]*

- Consider the following language

$$L = \{ wcw \mid w \in (a \mid b)^* \}$$

- These are some of the words in the language:
  - aabcaab
  - aca
  - bcb
- We can consider it an abstraction of the language that requires that identifiers must be declared before its use (before the "c" we would have the declaration of the variable, and after the "c" we would find its use):
  - In *aab*c*aab* the identifier would be *aab*
  - In *a*c*a* the identifier would be *a*
  - In *b*c*b* the identifier would be *b*
- This is a very simplified example of the real situation described (the variable might be used many times, for instance).

4

Language features which are not context-independent

Declaring identifiers before its use *[Aho]*

- It can be proved, using the **pumping lemma**, that the previous language is not context-independent.
- Any real programming language in which variables have to be declared before their use, therefore, will NOT be context-independent.
- Possible solutions:
  - Use a formalism more expressive than context-independent grammars, and an architecture which is more powerful than pushdown automata.
  - Add to the grammar a mechanism which allows us to solve the problem:
    - The symbols table
    - It is the most usual procedure.

5

Language features which are not context-independent

Declaring identifiers before its use *[Aho]*

- Let's consider the following language

$$L=\{a^n b^m c^n d^m \mid n\geq 1,\ m\geq 1\}$$

- If we analyse any of the words in this language…
  - *aaa***b***ccc***d**
  - *a***bb***c***dd**
  - *aaaaa***bb***ccccc***dd**
- The coincidence of the number of repetitions of the sequences of a's and c's can be interpreted as, for instance, checking that the number of parameters, when calling a subroutine, is the same as when the subroutine was declared.
  - In *aaa***b***ccc***d** the first function has been declared with three arguments (three a's), and then called with three arguments (three c's) *aaa*···*ccc*··· In the same way, the second function is declared and called with 1 parameter ···**b**···**d**.
  - The other two examples are alike.
- Observe that this language is a big simplification with respect to the real situation described (e.g. no function name and parameter types).

6

Language features which are not context-independent

Declaring identifiers before its use *[Aho]*

- It can be proved, using the **pumping lemma**, that the previous language is not context-independent.
- Any real programming language in which procedures are called with parameters and it is necessary to check its number or type will NOT be context-independent.

- Possible solutions:
  - Use a formalism more expressive than context-independent grammars, and an architecture which is more powerful than pushdown automata.
  - Add to the grammar a mechanism which allows us to solve the problem:
    - The symbols table
    - It is the most usual procedure.

7

Context-independent structures in programming languages

Structured programming

- Consider the following fragment of the ASPLE grammar:

```
7 : <stm train> ::= <statement>
8 :               | <statement> ; <stm train>
     ...
22 : <cond stm> ::= if <exp> then <stm train> fi
23 :              | if <exp> then <stm train> else <stm train> fi
24 : <loop stm> ::= while <exp> do <stm train> end
25 :              | repeat <stm train> until <exp>
```

- These rules represent, respectively, the following structures for flow-control:
  - Block of sentences.
  - Conditional flow.
  - Loops.
- The existence of these rules show that the language is context-independent (and it is not a regular language).

  repeat^n … until^n

8

Context-independent structures in programming languages

Conclusions

- We will be required to use
  - Context-independent grammars
  - A symbols table

9

Syntax Analyser

Strategies for syntactic analysis

- The various techniques for parsing can be grouped in two main types:
- First approach (**top-down**):
  - Having the axiom, the grammar and the program (transformed into a sequence of syntactic units) as starting points,
    - Try with the different derivation options for each non-terminal that we might find.
    - Proceed simultaneously along the program and the leaves of the derivation tree that we are building when we find a coincidence between both.
  - Until
    - Either we have generated the whole program from the axiom, using the grammar rules (and, so, the program was syntactically correct).
    - Or we have tried every single possible option for each non-terminal symbol (including the axiom), and it is utterly impossible to obtain the source program from the grammar, so this must be syntactically wrong.

10

Syntax Analyser

Strategies for syntactic analysis

- Second approach (**bottom-up**):
  - Having the sequence of tokens returned by the morphological analyser, we start reading it trying to match it with the right-hand sides of the grammar rules. There might be several possibilities for this.
  - When this happens, we substitute the sequence of tokens with the non-terminal at the left-hand side of the rule, and continue the same analysis.
  - If, at any time, none of the sequences in the string corresponds to any of the right-hand sides of the rules, we backtrack and try a different substitution.
  - When every option has been systematically tried, and the analysis could not be completed, the program must be syntactically incorrect.
  - The other way of finishing the analysis is when we finally obtain the axiom of the grammar alone.
    - In this last case, the program is syntactically correct.

11

Bottom-up analysis

Introduction

- The input string will be scanned, looking for coincidence with the right-hand sides of the rules in the grammar.
- While the terminal symbols match with the right-hand sides of any rule, we advance along the input.
- When we find the complete right-hand side of a rule, it will be reduced, by substituting that right-hand side with its left-hand side. In this way, the string (that must be reduced to the axiom) changes reproducing the derivation tree of the string from the bottom upwards.

12

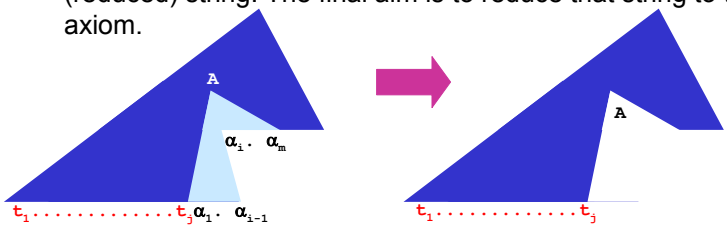
LR analysis

Concepts

- There are two main operations in bottom-up parsing: **shift** and **reduce**.
- Intuitively:
  - **Reduction**,
    - It occurs when all the components in the right-hand-side of a rule have all been matched with symbols in the input string.

$$A \rightarrow \alpha_1 \ldots \alpha_m$$

    - In essence, analysers replace the right-hand-side with the non-terminal symbol at the left of the rule, constructing the new (reduced) string. The final aim is to reduce that string to the axiom.

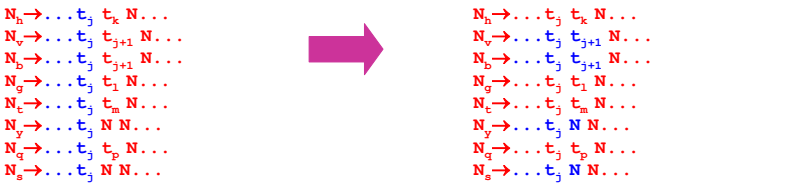


13

LR Analysis

Concepts

- Intuitively,
  - **Shift**,
    - It occurs when the terminals found in the string, one by one, match with the corresponding part of the right-hand side of a rule in the grammar.
    - The analysers note the circumstance and store the position in the right-hand side of all the rules that might finally be reduced in the input.
    - We shall see a few examples of this process.



$\ldots, t_j\ t_{j+1}\ t_{j+2}\ t_{j+3}\ t_{j+4}\ t_{j+5}\ t_{j+6}\ t_{j+7}, t_j, \ldots$ $\ldots, t_j\ t_{j+1}\ t_{j+2}\ t_{j+3}\ t_{j+4}\ t_{j+5}\ t_{j+6}\ t_{j+7}, t_j, \ldots$

14

LR Analysis

General LR algorithm: introduction

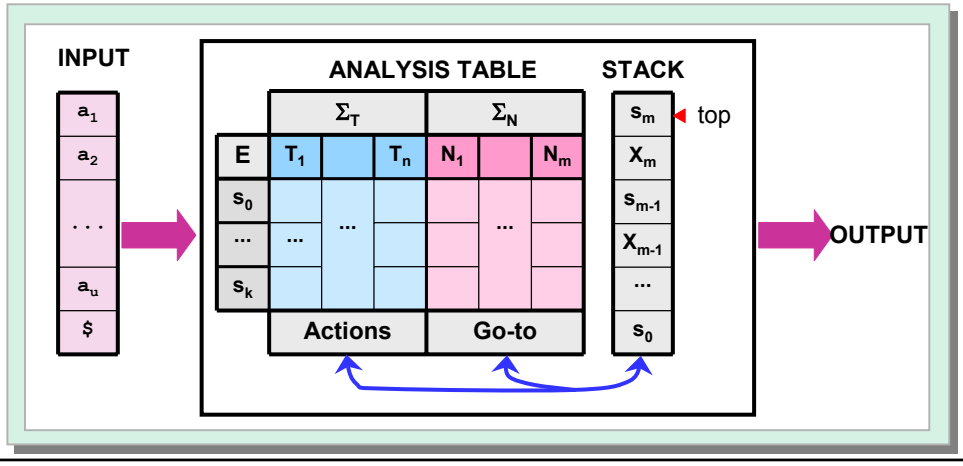
- This kind of analysis needs two steps:
  1. **Constructing** a **table** for syntactic analysis.
  2. **Using** that table in the analysis.
- Step 2 will be the same for all subtypes of the LR algorithm. These differ in step 1.
- The table has the following structure:
  - As many **columns** as **symbols** in the grammar (terminals and non-terminals).
    - The columns corresponding to terminal symbols determine the action that the parser does in each case.
    - The columns corresponding to non-terminals determine a "go-to" function.
  - As many **rows** as **analysis states** (seen afterwards).

15

LR Analysis

General LR algorithm: use of the table

- **Use of the table in the analysis**
  - An LR parser is a pushdown automata associated to a context-independent grammar, which has been modified so as to simplify as much as possible its coding, management and performance.
  - The following figure shows the structure of the parser:



16

LR Analysis

General LR algorithm: input, stack and table

- The analyser has the following components:
  - The **input** has, initially, the string that we want to analyse, followed by an ending symbol, e.g. "$"
  - The **stack**, where symbols will be introduced in pairs (at most), in the order $X_m s_m$
    - where X_m is terminal or non-terminal, and
    - s_m is a state
  - The **table**, which contains
    - **In the action columns,** instructions about how the analyser should perform if it is in the state corresponding to the row, and it reads the symbol indicated by the column. Possible actions are:
      - **sS**, shift the state "s"
      - **rP**, reduce using the production rule "P"
      - **Accept**, for accepting the analysis
      - **Error**, for ending the analysis with an error.
    - **In the "go-to" columns,** the transitions between the states. Therefore, the table cells contain names of states.
- Its behaviour is as follows:

17

LR Analysis

General LR algorithm: introductory example

- Let's consider the following grammar:

G_3 =<{E,T},
{i,+,(,)}
{E→ E+T
| T
T→ i
| (E)},
E>

18

LR Analysis

General LR algorithm: introductory example

- The grammar will be extended with a new axiom, whose only purpose is to add the end-of-string symbol:

$G_3 = \langle \{E, T, E'\}, \{i, +, (,), \$\}, \{E' \rightarrow E\$ \quad E \rightarrow E+T \mid T \quad T \rightarrow i \mid (E)\}, E' \rangle$

19

LR Analysis

General LR algorithm: introduction

- Let us assume that the analysis table is this (we shall see how to build it later):

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	s1	(*)	s2			4	3
1		r3		r3	r3		
2	s1		s2			5	3
3		r2		r2	r2		
4		s6			acc		
5		s6		s8			
6	s1		s2				7
7		r1		r1	r1		
8		r4		r4	r4		
	Action					Go-to	

20

LR Analysis

Algorithm of the analyser

- The algorithm can be summarised as follows:

```
State LRAnalyser(analysis_table, input, stack, grammar)
/* input contains the string "w$" to be analysed */
{
  pointer current_symbol=entrada[0];
  state  current_state;
  push(stack,0);
  while( true ) /* Unending loop */
  {
        if (analysis_table[current_state, current_symbol] == ss' ){
                push(stack, current_symbol);
                push(stack, s');
                current_symbol++;}
        else if ( analysis_table[current_state, current_symbol] == rj ){
        /* Assume the j-th rule is A→α */
                "perform 2*longitud(α) pop(stack)"
                s'←top(stack);
                push(A);
                push(stack, analysis_table[s',A]));
                printf("Reduce: A→α");}
        else if ( analysis_table[current_state, current_symbol] == accept )
                return "ACCEPTED STRING";
        else /* empty cell */
                return "Syntactic error: REJECTED STRING");
  }
}
```

21

LR Analysis: introductory example

(0) E'→E
(1) E → E+T
(2) | T
(3) T → i
(4) | (E)

i	+	i	+	i	$	
0						

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	s1		s2			4	3
1		r3		r3	r3		
2	s1		s2			5	3
3		r2		r2	r2		
4		s6			acc		
5		s6		s8			
6	s1		s2				7
7		r1		r1	r1		
8		r4		r4	r4		
	Action					Go-to	

22

LR Analysis: introductory example

```
(0) E'→E
(1) E → E+T
(2)   |   T
(3) T → i
(4)   |   (E)
```

i	+	i	+	i	$	
1	i	0				

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	s1		s2			4	3
1		r3		r3	r3		
2	s1		s2			5	3
3		r2		r2	r2		
4		s6			acc		
5		s6		s8			
6	s1		s2				7
7		r1		r1	r1		
8		r4		r4	r4		
	Action					Go-to	

23

LR Analysis: introductory example

```
(0) E'→E
(1) E → E+T
(2)   |   T
(3) T → i
(4)   |   (E)
```

i	+	i	+	i	$	
3	T	0				

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	s1		s2			4	3
1		r3		r3	r3		
2	s1		s2			5	3
3		r2		r2	r2		
4		s6			acc		
5		s6		s8			
6	s1		s2				7
7		r1		r1	r1		
8		r4		r4	r4		
	Action					Go-to	

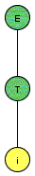
24

LR Analysis: introductory example

```
(0) E'→E
(1) E → E+T
(2)   |   T
(3) T → i
(4)   |   (E)
```

i	+	i	+	i	$	
4	E	0				

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	s1		s2			4	3
1		r3		r3	r3		
2	s1		s2			5	3
3		r2		r2	r2		
4		s6			acc		
5		s6		s8			
6	s1		s2				7
7		r1		r1	r1		
8		r4		r4	r4		
	Action					Go-to	



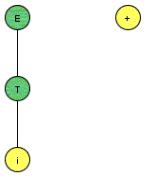
25

LR Analysis: introductory example

```
(0) E'→E
(1) E → E+T
(2)   |   T
(3) T → i
(4)   |   (E)
```

i	+	i	+	i	$	
6	+	4	E	0		

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	s1		s2			4	3
1		r3		r3	r3		
2	s1		s2			5	3
3		r2		r2	r2		
4		s6			acc		
5		s6		s8			
6	s1		s2				7
7		r1		r1	r1		
8		r4		r4	r4		
	Action					Go-to	



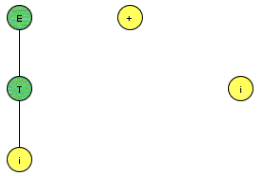
26

LR Analysis: introductory example

```
(0) E'→E
(1) E → E+T
(2)   |   T
(3) T → i
(4)   |   (E)
```

i	+	i	+	i	$	
1	i	6	+	4	E	0

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	s1		s2			4	3
1		r3		r3	r3		
2	s1		s2			5	3
3		r2		r2	r2		
4		s6			acc		
5		s6		s8			
6	s1		s2				7
7		r1		r1	r1		
8		r4		r4	r4		
	Action					Go-to	



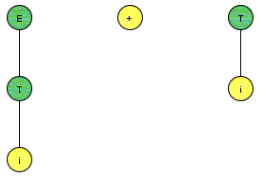
27

LR Analysis: introductory example

```
(0) E'→E
(1) E → E+T
(2)   |   T
(3) T → i
(4)   |   (E)
```

i	+	i	+	i	$	
7	T	6	+	4	E	0

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	s1		s2			4	3
1		r3		r3	r3		
2	s1		s2			5	3
3		r2		r2	r2		
4		s6			acc		
5		s6		s8			
6	s1		s2				7
7		r1		r1	r1		
8		r4		r4	r4		
	Action					Go-to	



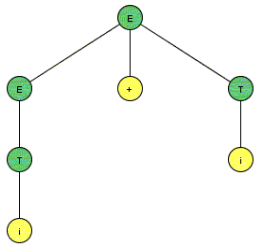
28

LR Analysis: introductory example

```
(0) E'→E
(1) E → E+T
(2)   |   T
(3) T → i
(4)   |   (E)
```

i	+	i	+	i	$	
4	E	0				

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	s1		s2			4	3
1		r3		r3	r3		
2	s1		s2			5	3
3		r2		r2	r2		
4		s6			acc		
5		s6		s8			
6	s1		s2				7
7		r1		r1	r1		
8		r4		r4	r4		
	Action					Go-to	



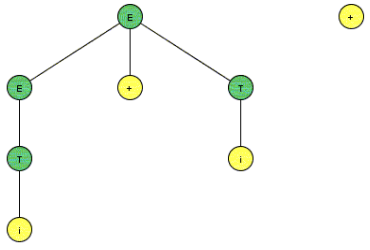
29

LR Analysis: introductory example

```
(0) E'→E
(1) E → E+T
(2)   |   T
(3) T → i
(4)   |   (E)
```

i	+	i	+	i	$	
6	+	4	E	0		

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	s1		s2			4	3
1		r3		r3	r3		
2	s1		s2			5	3
3		r2		r2	r2		
4		s6			acc		
5		s6		s8			
6	s1		s2				7
7		r1		r1	r1		
8		r4		r4	r4		
	Action					Go-to	



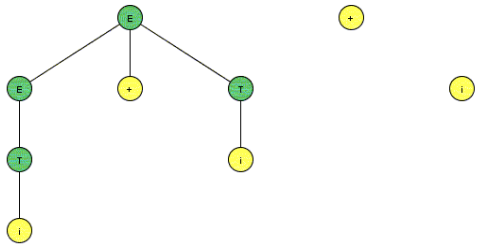
30

LR Analysis: introductory example

```
(0) E'→E
(1) E → E+T
(2)   |   T
(3) T → i
(4)   |   (E)
```

i	+	i	+	i	$	
1	i	6	+	4	E	0

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	s1		s2			4	3
1		r3		r3	r3		
2	s1		s2			5	3
3		r2		r2	r2		
4		s6			acc		
5		s6		s8			
6	s1		s2				7
7		r1		r1	r1		
8		r4		r4	r4		
	Action					Go-to	



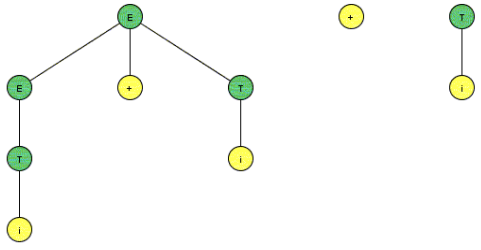
31

LR Analysis: introductory example

```
(0) E'→E
(1) E → E+T
(2)   |   T
(3) T → i
(4)   |   (E)
```

i	+	i	+	i	$	
7	T	6	+	4	E	0

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	s1		s2			4	3
1		r3		r3	r3		
2	s1		s2			5	3
3		r2		r2	r2		
4		s6			acc		
5		s6		s8			
6	s1		s2				7
7		r1		r1	r1		
8		r4		r4	r4		
	Action					Go-to	



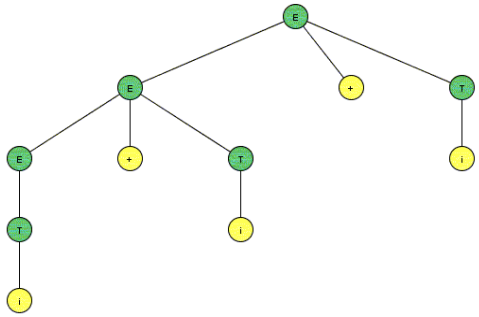
32

LR Analysis: introductory example

(0) E'→E
(1) E → E+T
(2) | T
(3) T → i
(4) | (E)

i	+	i	+	i	$	
4	E	0				

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	s1		s2			4	3
1		r3		r3	r3		
2	s1		s2			5	3
3		r2		r2	r2		
4		s6			acc		
5		s6		s8			
6	s1		s2				7
7		r1		r1	r1		
8		r4		r4	r4		
	Action					Go-to	



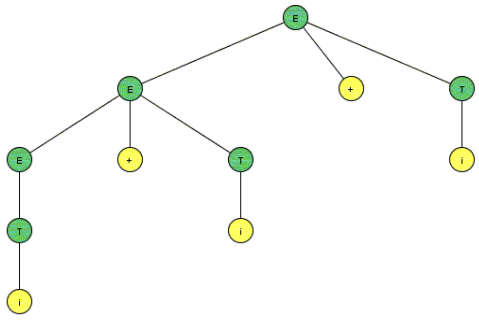
33

LR Analysis: introductory example

(0) E'→E
(1) E → E+T
(2) | T
(3) T → i
(4) | (E)

i	+	i	+	i	$	
E	0					

	Σ_T					Σ_N	
E	i	+	(	)	$	E	T
0	s1		s2			4	3
1		r3		r3	r3		
2	s1		s2			5	3
3		r2		r2	r2		
4		s6			acc		
5		s6		s8			
6	s1		s2				7
7		r1		r1	r1		
8		r4		r4	r4		
	Action					Go-to	



34

LR Analysis: exercise

Problem

- Repeat the analysis performed by the previous example, using the analysis table and input string shown in the next page.

35

LR Analysis: example

```
state AnalizadorLR(
```

	Σ_T						Σ_N		
E	I	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6				acc			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			
	Action						Go-to		

id	*	id	+	id	$	

```
(1) E → E+T
(2)   |   T
(3) T → T*F
(4)   |   F
(5) F →(E)
(6)   |   id
```

)

36