

CS 333 Compiler Design

Dr. Anup Keshri (anup_keshri@bitmesra.ac.in)

Dr. Supratim Biswas (supratim.biswas@bitmesra.ac.in)

Department of Computer Science and Engineering
BIT Mesra

Course Information

- Read the basic course information : Course Objectives, Course Outcomes, CO-PO-PSO mapping; syllabus, from the home page.

Course Objectives : 1. understand the need for compiler in computer engineering

2. provide a thorough understanding of design, working and implementation of PL

3. trace the major concept areas of language translation and compiler design

4. create awareness of functioning & complexity of modern compilers

Course Information

- Course Outcomes :
1. analyze need for compiler for interfacing between user and m/c
 2. Explain roles of several phases of compilation process
 3. Create awareness of functioning / complexity of modern compilers
 4. Outline major concept areas of language translation / compiler design
 5. Develop a comprehensive compiler for a given language
 6. Apply knowledge for developing tools for NLP

Course Conduct

- 4 contact hrs / week : 3 lectures and 1 tutorial
- Prescribed Evaluation scheme will be followed
- Prepare for fill in the blanks type of questions in quizzes
- Attendance norms of the institute apply – do not approach the instructors for relaxation in attendance
- Direct positive correlation observed in FLAT between active presence and skill enhancement / better performance
- Course and the associated laboratory course CS 334 will be synchronized to reinforce the concepts and skills.

Distinctive Features of a Compilers Course

- Unique course in CSE discipline. It bridges deep theoretical concepts to the most recent advances in architectures and operating systems
- Awareness of compiler features helps in increased programming productivity (invest time and effort to know the capabilities of your compiler)
- Two drivers for Research in compilers - design of new programming languages and invention of new architectures.
- Research has produced spectacular enhancements in technology. Designing a working compiler 3 decades back required several human years.
- Today, given m different languages and n different architectures, $m \times n$ compilers can be generated in hours / days.
- Research in compilers continues, specially for automatic optimization, automatic parallelization of sequential programs.
- The huge leaps in hardware technology have not been efficiently exploited till date because software tools have not been able to catch up with the native computational powers of contemporary High Performance Architectures.

Complexity of Software Systems

The following table gives an idea about the size and complexity of large software systems from 3 highly popular domains.

The information is only meant to be indicative as the figures are significantly larger today.

Name	Category	1 st Rel	No of Dirs	No of Files	Lines of Code	White spaces	No of Langs	Main Language
Linux 4.9-rc.5	OS Kernel	1991	3830	45884	20659032	36%	18	C
Gcc 6.2.0	Compiler Framework	1987	5500	79499	10638200	33%	33	C, C++
Mysql 5.7.16	DBMS	1995	1533	10656	3901791	29%	27	C++, C

What is a Compiler ?

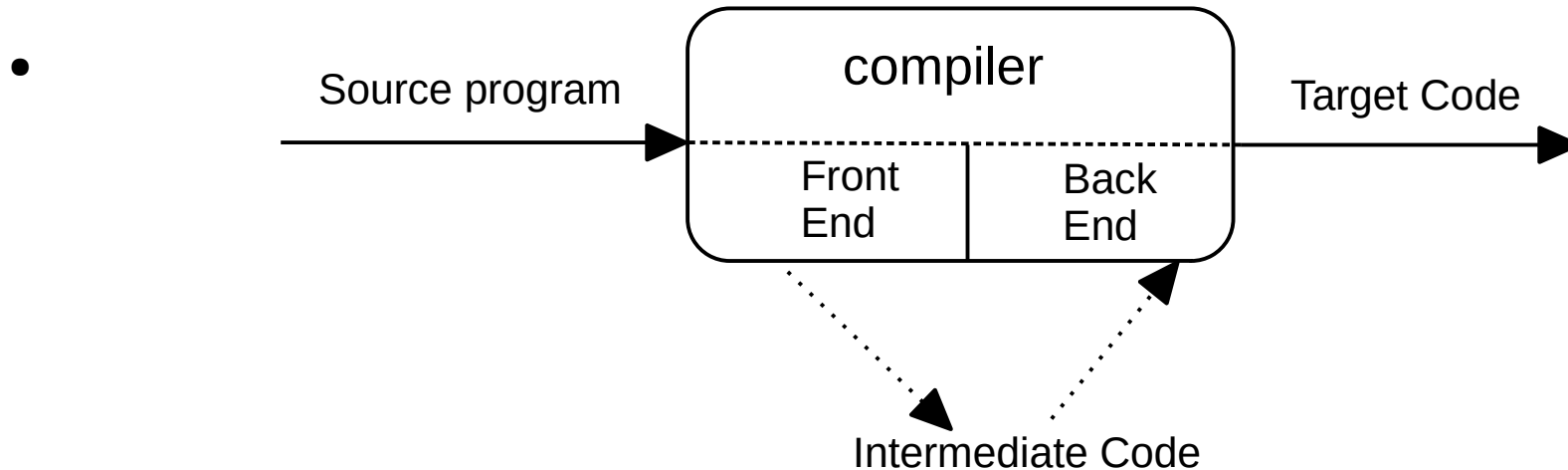
- A compiler is essentially a Translator from a High Level Language (HLL) such as C, C++, Java to a Low Level Language (LLL) such as assembly or machine language.



The difference in the level of abstractions used by HLL and LLL are huge. HLLs focus on application domain computations while LLLs depend on the target architecture.

- The translation process used in a compiler is non-trivial. A compiler slowly brings down the HLL computations to instructions supported by an architecture.

Another view of a Compiler



- The front-end of a compiler performs HLL specific analyses and produces semantically equivalent code (intermediate code) whose abstraction level is significantly lower than HLL but still higher than LLL.
- The back-end of a compiler analyses the intermediate code produced by the front-end and is responsible for generating target code.

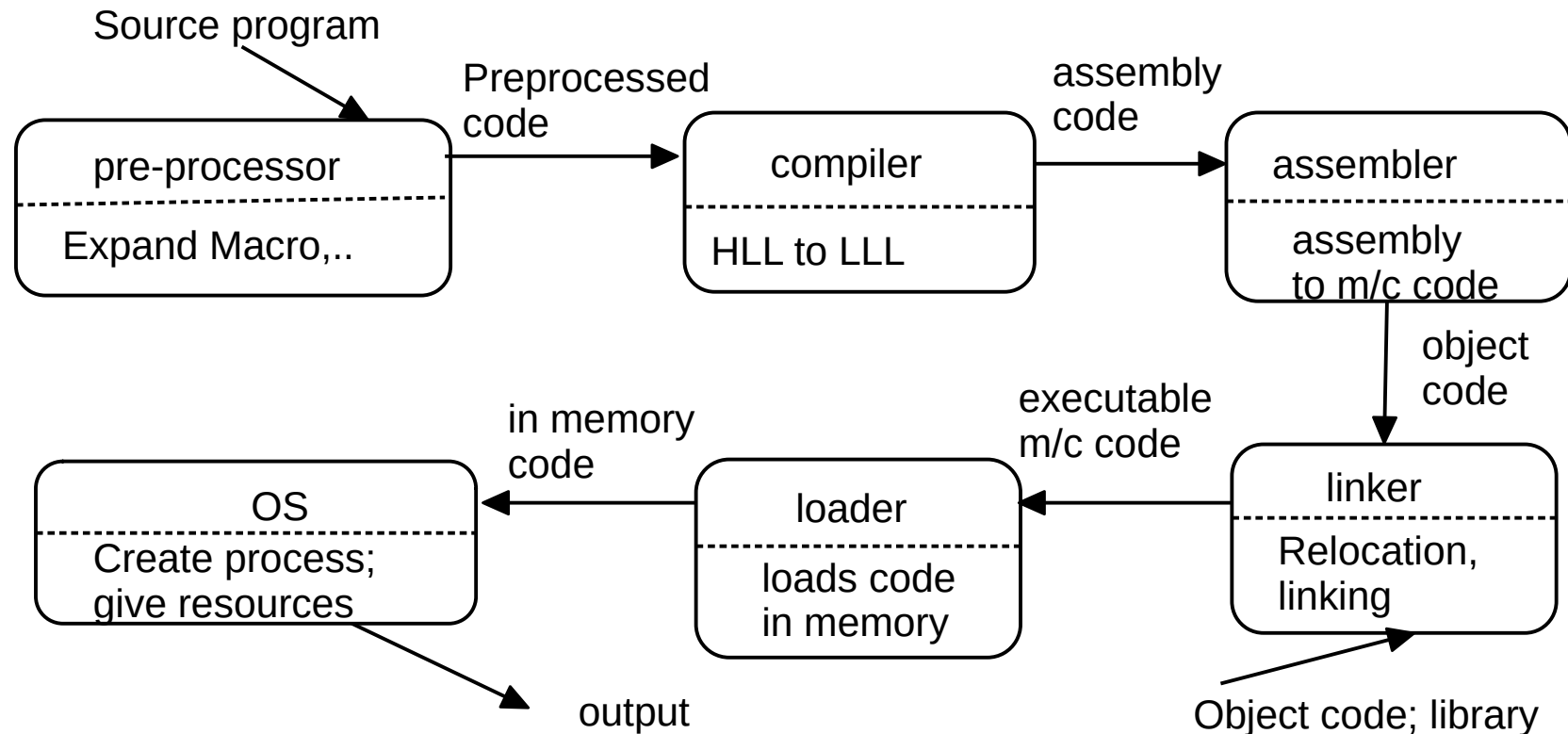
Compiler Phase and Pass

Phase and pass are two commonly used terms in compiler design.

- The process of compilation is carried out in distinct logical steps, which are called phases.
- A compiler typically has 7 phases : lexical analysis, syntax analysis, semantic analysis, intermediate code generation, run time environment, code optimization and code generation
- A pass denotes a processing of the entire source code or its equivalent form. Industrial quality compilers use large number of passes in their translation process.

Compiler and Support Software

A compiler is not an end to end software. It uses several system software tools to accomplish its goal.



Other Models of Translation

A compiler is not the only way to translate code from HLL to LLL.

- Another mechanism, known as Intrepretation, is also used popularly. Such software are called Interpreters.
- In principle a HLL may be either compiled or interpreted, but in practice some languages are compiled and some are interpreted or same use both the models.
- Difference between Compiler and Interpreter
- Identify languages that are interpreted and languages that are compiled.

THEME 1

COMPILERS : EXAMPLE DRIVEN APPROACH

Simple C program

```
#include <stdio.h>

int main()
{   int a[1000], i, j;
    int sum = 10000;
    for (i = 0; i < 1000; i++) a[i] = i;
    for (i = 0; i < 1000; i++)
        for (j = 0; j < i*i; j++)
            a[i] = a[i] + a[j];
    printf(" sum : %d \n", sum);
}
```

C program – Manual Analysis

Let us manually analyse “firstprog.c”

- What is the role played by the statement ? `#include <stdio.h>`
- Someone has to supply the prototype of the i/o function `printf()` in order to check whether the call is valid
- Let us assume there exists a s/w program that will do this task.
- In real world this is task of a tool called C pre-processor, named as “cpp” which is called by C compiler when it encounters “`#include...` ” and similar other statements.
- For the manual processing we shall ignore this statement for the present.

C program – Manual Analysis

The remaining statements in “firstprog.c”

- The pretty indented display of the text in the earlier slide was due to an editor which interpreted certain characters before displaying.
- The contents of the raw text in the file is shown below, where the symbol, \leftarrow represents newline and \leftrightarrow denotes a single white space. This is how a compiler sees the program at the first instance.

```
 $\leftarrow$ int  $\leftrightarrow$  main() $\leftarrow$ { $\leftarrow$ int  $\leftrightarrow$   $\leftrightarrow$  a[1000],  $\leftrightarrow$  i,  $\leftrightarrow$  j; $\leftarrow$   $\leftrightarrow$   $\leftrightarrow$  int  
 $\leftrightarrow$  sum  $\leftrightarrow$  =  $\leftrightarrow$  10000; $\leftarrow$   $\leftrightarrow$   $\leftrightarrow$  for  $\leftrightarrow$  (i  $\leftrightarrow$  =  $\leftrightarrow$  0;  $\leftrightarrow$  i  $\leftrightarrow$  <  $\leftrightarrow$  1  
000;  $\leftrightarrow$  i++)  $\leftrightarrow$  a[i]  $\leftrightarrow$  =  $\leftrightarrow$  i; $\leftarrow$  .....
```

The first task is then to break the stream of characters shown above into meaningful units of language C.

Basic Elements of C program

Breaking the program string into meaningful words of the language give the smallest logical units of the language.

- Why is “main” a word and not “main()” is decided by the programming language (PL) specifications and not by a compiler.
- The elements in the red boxes are called tokens (lexemes) and a compiler first converts the character stream into a sequence of tokens.

- | | | | | | | | | | |
|-----|------|---|---|---|-----|-----|-----|---|-------|
|] | , | i | , | j | ; | int | sum | = | 10000 |
| ; | for | (| i | = | 0 | ; | i | < | 1000 |
|] | , | i | , | j | ; | int | sum | = | 10000 |
| int | main | (|) | { | int | a | [| | 1000 |

Lexical Analysis

- The process of partitioning a program into its constituent smallest logical words is called as Lexical Analysis.

- Distinction between lexemes and tokens

- Lexeme : instance of the smallest word

Examples : int main a sum for 1000 < ++ * =

- Token : groups of similar lexemes

Examples : identifier = {i j a sum }

keyword = {main int for }

integer = {0 1000 10000}

operator = {< ++ * = }

Lexical Analysis

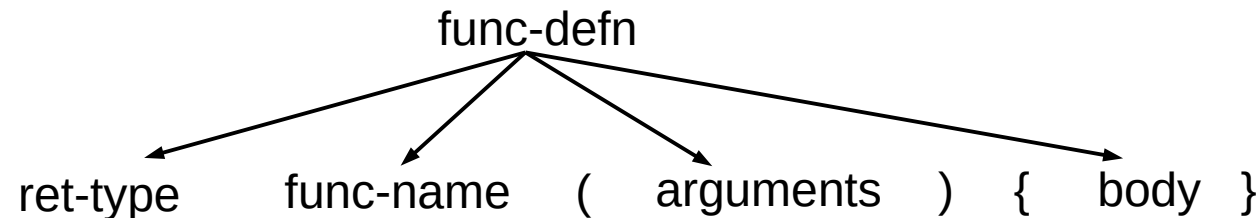
Lexemes Tokens and Patterns

- How to describe the lexemes that form the token identifier
- Different PL have variations in the specification of an identifier :
 - string of alphanumeric characters with an alphabet as the first character
 - string of alphanumeric characters with an alphabet as the first character but length is restricted to 31 characters
 - string of alphabet or numerals or underline (`_`) characters with an alphabet or underline as the first character with length ≤ 31 ; characters beyond 31 are ignored
- Descriptions as given above, may be formal or informal are called as patterns. A pattern when described formally gives a precise description of the underlying token. We are already familiar with the formal notation of regular expression which is used to describe patterns.
- The benefit of formal pattern description is that one can directly design a recognizer (DFA) from the patterns.

Structure of a Program

Lexical analysis partitions an input program into a stream of tokens (reduces the size of the input).

- The immediate next task is to determine if the consecutive tokens, when grouped together, describe some structure of the PL.
- A PL defines the linguistic structures that constitute a program in the language. For example, a C program is a collection of functions and a function has a hierarchical structure as shown below.
- The structure of func-defn is defined using sub-structures, ret-type, func-name, arguments and body, as shown below.

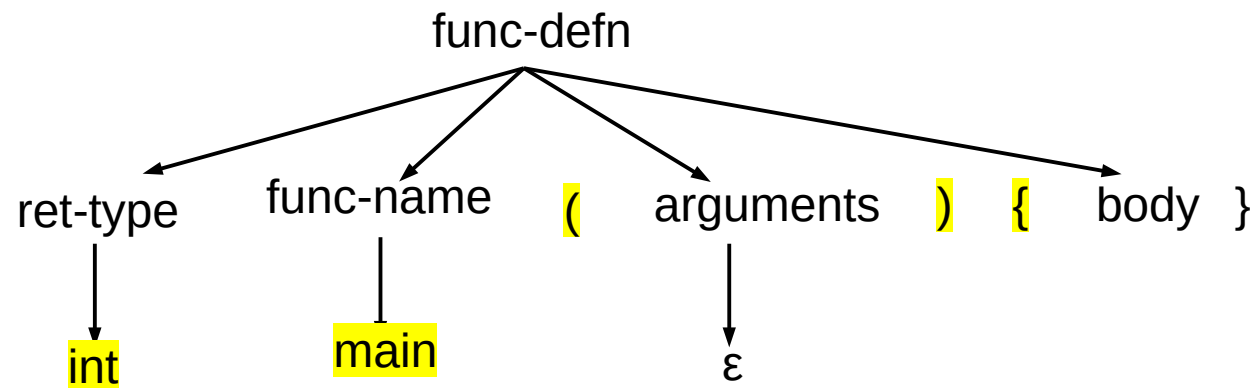


Structure of a Program

Let us intuitively apply the ideas to the C program to discover its structure.

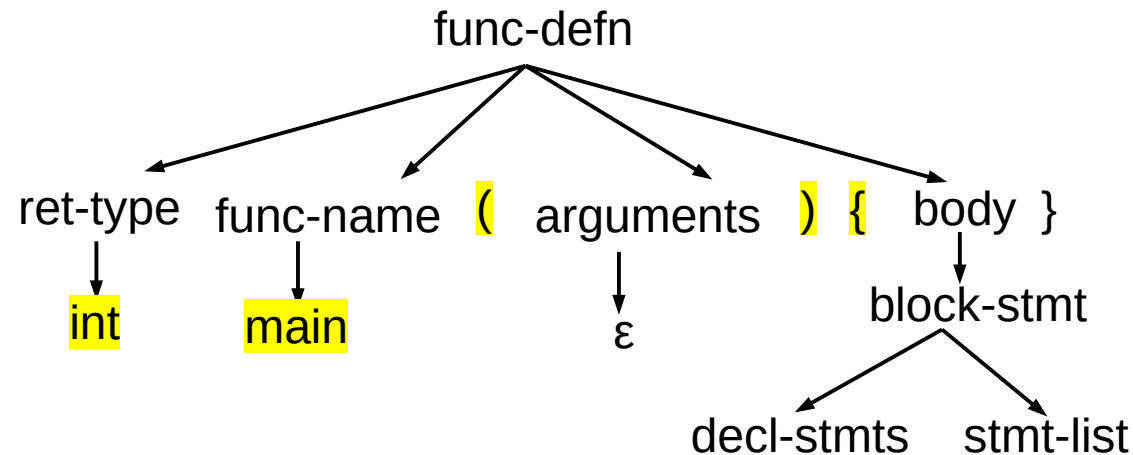
The input “int main() {“

partially matches the program structure as displayed in the tree below.



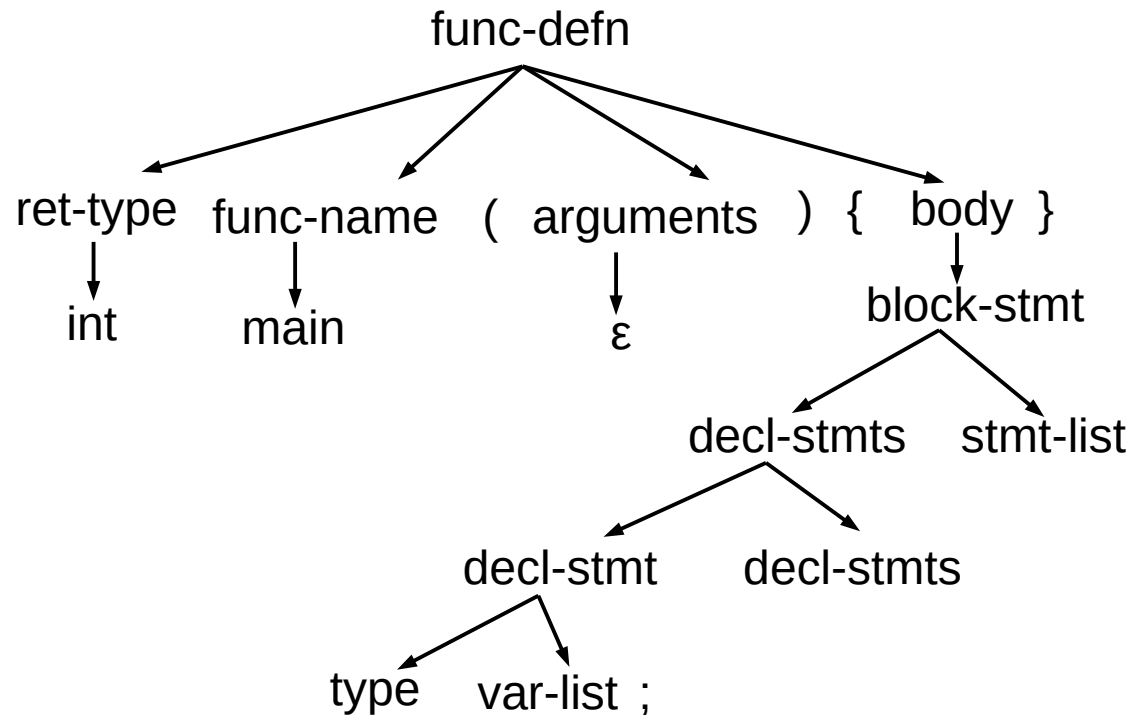
Structure of a Program

The structure of a function body is further elaborated into a block-statement which in turn is specified by one or more declaration statements followed by a list of statements.



Structure of a Program

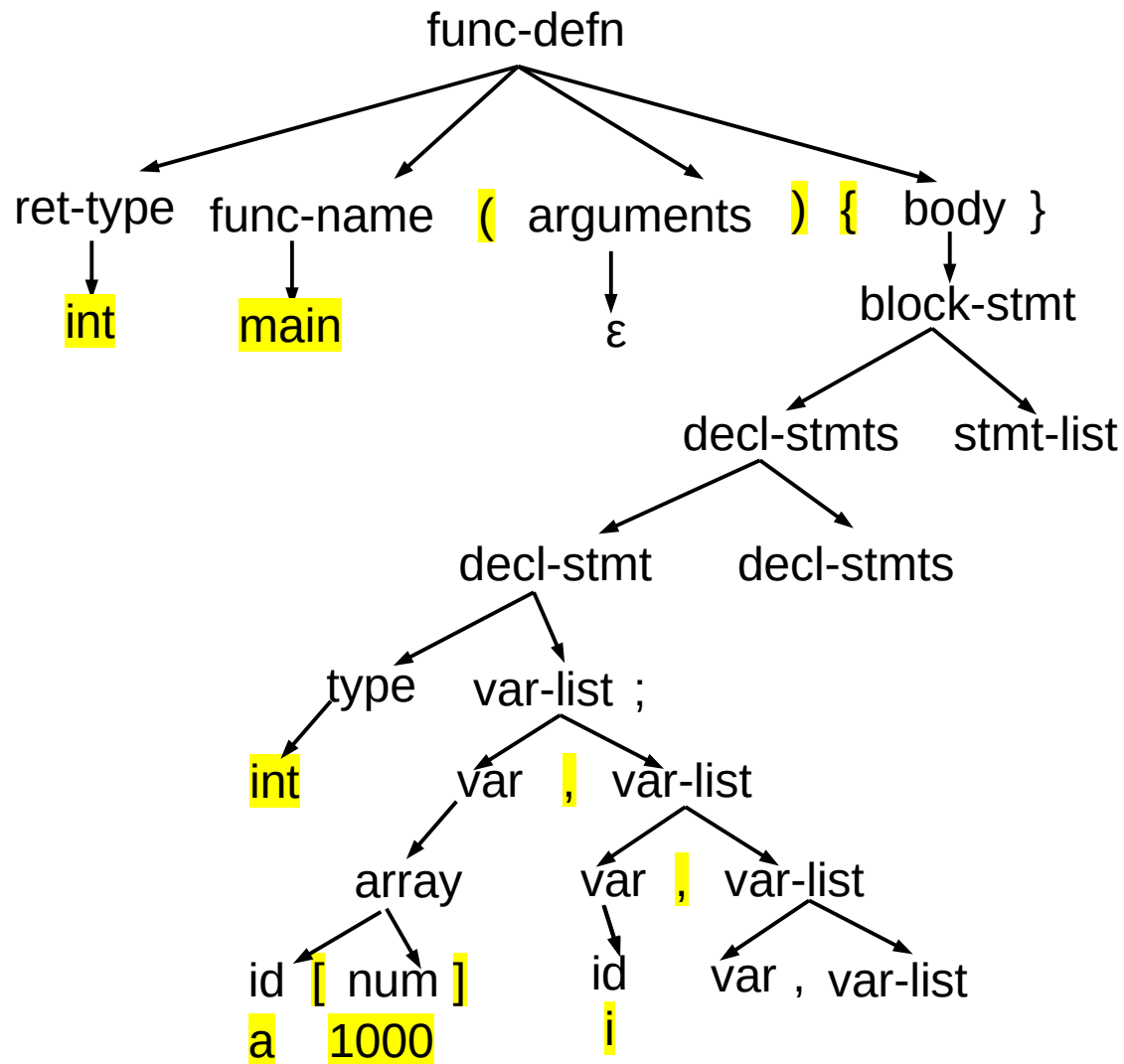
The structure of a single declaration statement is used to further extend the tree under decl-stmts – which comprises of a type specifier followed by a list of variable declarations.



The tree gets extended when the input examined is :

`int main () { int a [1000] , i ,` is shown in the next page.

Structure of a Program



Tree for Validating Program Structure

- The process of discovering language constructs in a stream of input tokens is called **Syntax Analysis**. Syntax analysis uses lexical analysis to convert lexemes to tokens and works on tokens only.
- A tree is a natural data structure to represent the structure of a program and is constructed incrementally as input tokens are examined one by one.
- Such a tree, often called a Parse Tree or an Abstract Syntax Tree (AST), is used for representation of discovered programming language structures.
- The tokens lie at the leaf nodes of the tree while the internal nodes are the names of language features rooted at this node.
- The structural linguistic features have to be clearly specified to the compiler designer.
- A CFG meets the requirement for specifying the syntactic aspects of language constructs.

CFG for Syntax Specification

- As an illustration, we write a CFG underlying the structure validation process
- $G = (N, T, P, S)$ where $N = \{ \text{func-defn, ret-type, func-name, arguments, body, block-stmt, decl-stmts, stmt-list, decl-stmt, type, var-list, var, array} \}$
- T has the following symbols and tokens : $() \{ \} \text{int} , ; \text{id} [] \text{num}$
- The nonterminal func-defn is the start symbol S
- The production rules follow :

$\text{func-defn} \rightarrow \text{ret-type func-name (arguments) \{ body \}}$

- $\text{ret-type} \rightarrow \text{int} \mid \text{void} \mid \epsilon$
 $\text{arguments} \rightarrow \text{arg-list} \mid \epsilon$
 $\text{block-stmt} \rightarrow \text{decl-stmts stmt-list}$
 $\text{decl-stmt} \rightarrow \text{type var-list ;}$
 $\text{var} \rightarrow \text{array} \mid \text{id}$
- $\text{func-name} \rightarrow \text{id}$
 $\text{body} \rightarrow \text{block-stmt} \mid \epsilon$
 $\text{decl-stmts} \rightarrow \text{decl-stmt} \mid \epsilon$
 $\text{var-list} \rightarrow \text{var} , \text{varlist}$
 $\text{array} \rightarrow \text{id [num]}$

Understanding the Meaning

Semantic Analysis

Semantic Analysis

Semantic Analysis

Intermediate Code Generation

Intermediate Code Generation

Intermediate Code Generation

Intermediate Code Generation

Intermediate Code Generation

Run Time Environment

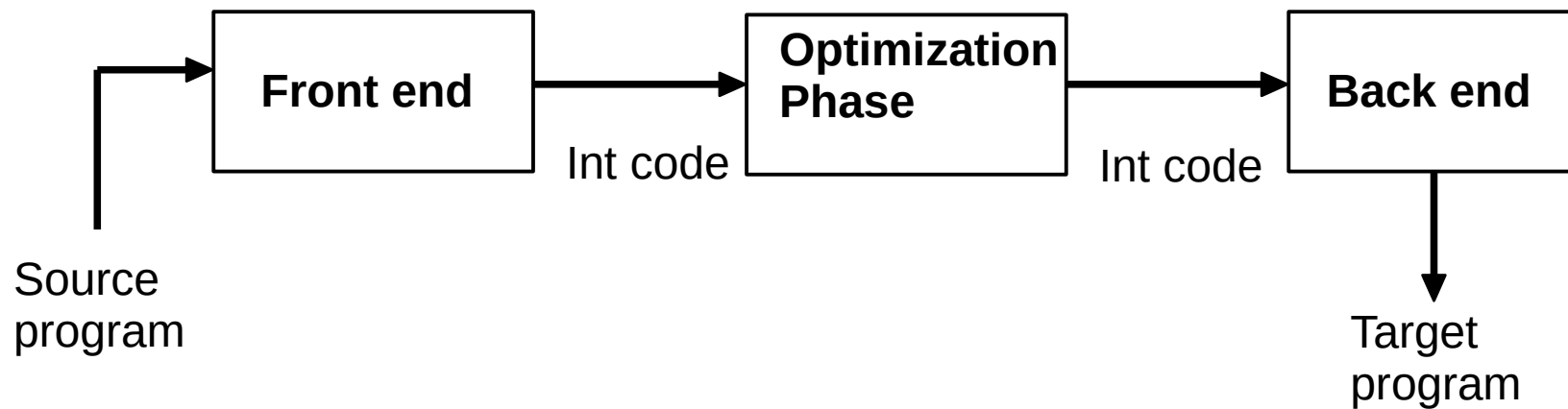
Run Time Environment

Run Time Environment

Run Time Environment

Compiler Optimizations

- Intermediate code optimization is an optional phase which is enabled by an user if so required.



- Purpose is to improve the quality of intermediate code generated by the front end of the compiler.
- Illustrate the capability of this phase through examples.

Compiler Optimizations

- Consider the following C program.

```
int main()
{  int a[1000], i, j;
    int sum = 10000;
    for (i = 0; i < 1000; i++) a[i] = i;
    for (i = 0; i < 1000; i++)
        for (j = 0; j < i*i; j++)  a[i] = a[i] + a[j];
    printf(" sum : %d \n", sum);
}
```


Compiler Optimizations

- Compile this program with

```
$ gcc -S -fverbose-asm firstprog.c
```

```
int main()
{  int a[1000], i, j;
    int sum = 10000;
    for (i = 0; i < 1000; i++) a[i] = i;
    for (i = 0; i < 1000; i++)
        for (j = 0; j < i*i; j++)  a[i] = a[i] + a[j];
    printf(" sum : %d \n", sum);
}
```

Compiler Optimizations

- Compile this program with
`$ gcc -S -fverbose-asm firstprog.c -o unopt.s`
- Now compile using the O2 switch
`$ gcc -S -fverbose-asm firstprog.c -O2 -o opt.s`
- Compare the two assembly code side by side and observe.
- You may be surprised by the second assembly code.

Manual Optimization

Consider the following C++ program :

```
int main()
{ int a = 2, b = 3, c = 40, d, i, j;
  int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
  for ( i = 1; i < 11; i+=2 )
  { if ( i%2) { d = a * b; x[i] = x[i-1] + d; c = b*100; }
    else {d = a * a; x[(i+d)%10] = x[i]+x[i-1];} };
  if (a < 2) for (j = 1; j < 11; j++) x[(j+5)%10] = x[j] + 5;
  else for (j = 0; j < 10; j++) cout << x[j] << " ";
  cout << endl;
  return 0;
}
```

Constant Evaluation

Task 1 : Evaluate compile time constants. What are the savings ?

```
int main()
{ int a = 2, b = 3, c = 40, d, i, j; int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
  for ( i = 1; i < 11; i+=2 )
  { if ( i%2)
    { d = 6; // d = a * b;
      x[i] = x[i-1] + d;
      c = 300; // c = b*100;
    }
    else { d = 4; // d = a * a;
          x[(i+d)%10] = x[i]+x[i-1];} };
  if (False) for (j = 1; j < 11; j++) x[(j+5)%10]=x[j]+5; // 2 < 2
  else for (j = 0; j < 10; j++) cout << x[j] << " "; cout << endl;
  return 0;
}
```

Constant Propagation

Task 2 : Evaluate compile time constants. What are the savings ?

```
int main()
{ int a = 2, b = 3, c = 40, d, i, j; int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
  for ( i = 1; i < 11; i+=2 )
  { if ( i%2)
    { d = 6; // d = a * b;
      x[i] = x[i-1] + 6;      // substitute 6 for d
      c = 300; // c = b*100;
    }
    else { d = 4; // d = a * a;
          X [ (i+4) % 10] = x[i] + x[i-1];} };
  if (False) for (j = 1; j < 11; j++) x[(j+5)%10]=x[j]+5;    // 2 < 2
  else for (j = 0; j < 10; j++) cout << x[j] << " "; cout << endl;
  return 0;
}
```

Loop Invariant Code Motion

Task 3 : Move loop invariant code out of loops. What are the savings ?

```
int main()
{ int a = 2, b = 3, c = 40, d, i, j; int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
  d = 6; // d = a * b;
  c = 300; // c = b*100;
  d = 4; // d = a * a;
  for ( i = 1; i < 11; i+=2 )
  { if ( i%2)
      { x[i] = x[i-1] + 6; }
    else { x[(i+4) % 10] = x[i] + x[i-1];} };
  if (False ) for (j = 1; j < 11; j++) x[(j+5)%10]=x[j]+5; // 2 < 2
  else for (j = 0; j < 10; j++) cout << x[j] << " "; cout << endl;
  return 0;
}
```

Live Variables & Dead Code Removal

Task 4 : Find variables that are live and remove dead code. What are the savings?

```
int main()
{ int a = 2, b = 3, c = 40, d, i, j;
  int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
  for ( i = 1; i < 11; i+=2 )
  { if ( i%2)
      { x[i] = x[i-1] + 6; }
    else { x[(i+4) % 10] = x[i] + x[i-1];} };
  if (False ) for (j = 1; j < 11; j++) x[(j+5)%10]=x[j]+5;    // 2 < 2
    else for (j = 0; j < 10; j++) cout << x[j] << " "; cout << endl;
  return 0;
}
```

Remove Unreachable Code

Task 5 : Find variables that are unreachable, evaluate conditional expressions in control flow constructs and remove unreachable code. What are the savings?

```
int main()
{ int a = 2, b = 3, c = 40, d, i, j;
  int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
  for ( i = 1; i < 11; i+=2 )
  { if ( i%2)
      { x[i] = x[i-1] + 6; }
    else { x[(i+4) % 10] = x[i] + x[i-1];} };
    for (j = 0; j < 10; j++) cout << x[j] << " "; cout << endl;
  return 0;
}
```


Remove Unused Definitions

Task 6 : Find the definitions that are not used in the program. What are the savings?

```
int main()
{ // int a = 2, b = 3, c = 40, d,
  int i, j;
  int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
  for ( i = 1; i < 11; i+=2 )
  { if ( i%2)
      { x[i] = x[i-1] + 6; }
    else { x[(i+4) % 10] = x[i] + x[i-1];} };
    for (j = 0; j < 10; j++) cout << x[j] << " "; cout << endl;
  return 0;
}
```

Final Optimized Program

**Task 6 : Find the definitions that are not used in the program.
What are the savings?**

```
int main()
{ int i, j;
  int x[10]={10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
  for ( i = 1; i < 11; i+=2 )
  { if ( i%2)
      { x[i] = x[i-1] + 6; }
    else { x[(i+4) % 10] = x[i] + x[i-1];} };
    for (j = 0; j < 10; j++) cout << x[j] << " "; cout << endl;
  return 0;
}
```

Code Generation

Code Generation

Compiler Concepts Learned Through Example

THEME 2

STRUCTURE OF A COMPILER

Design of a Compiler

- Regular expressions: **Finite Automata : Lexical Analyser**
- Context free grammars : **Pushdown automata : Parser or Syntax Analyser**
- Nested Symbol Tables : Scope and lifetime analysis : **Data structure and algorithms**
- Memory layout and activation records : Runtime environments : **Architecture and OS**

Design of a Compiler

- Semantic Analysis [Context Sensitive Issues such as Type checking; intermediate code generation] : Syntax directed translation scheme : Tree, graph; **Data structure and algorithms.**
- Parameter passing mechanisms : Semantic Analyser : **Programming Languages.**
- Optimization of intermediate code : Optional pass : graph, lattice theory, solution of equations; **Discrete Structures.**
- Code generation : Compiler back-end design: **Algorithms and complexity; Architecture, Assembly language, OS.**

Theory and Practice are intimately connected

THEME 3

Experimentation With Gnu C Compiler
Open Source Production Quality Compiler

Simple C program

```
#include <stdio.h>

int main()
{   int a[1000], i, j;
    int sum = 10000;
    for (i = 0; i < 1000; i++) a[i] = i;
    for (i = 0; i < 1000; i++)
        for (j = 0; j < i*i; j++)
            a[i] = a[i] + a[j];
    printf(" sum : %d \n", sum);
}
```

Compiler Features

- Most compilers provide a host of features for programming ease and enhanced productivity.
- Illustrated with Gnu C compiler (gcc) & C++ compiler (g++)
- List a few compiler options that are generally useful. Read the online manual, [man gcc](#) or [info gcc](#) for more details.
- -fverbose-asm -S -c -o
-v -O2 -E
-fump-tree-all and many many others
- Experiments to illustrate the working of a compiler through a simple example.

Checking Correctness

The output produced after the 6 optimizations are performed in the order discussed are as follows.

10 16 30 36 50 56 70 76 90 96	original source
10 16 30 36 50 56 70 76 90 96	after optimization 1
10 16 30 36 50 56 70 76 90 96	after optimizations 1,2
10 16 30 36 50 56 70 76 90 96	after optimization 1 to 3
10 16 30 36 50 56 70 76 90 96	after optimization 1 to 4
10 16 30 36 50 56 70 76 90 96	after optimization 1 to 5
10 16 30 36 50 56 70 76 90 96	after optimization 1 to 6
10 16 30 36 50 56 70 76 90 96	after optimization by gcc

Compiling for Dumps

Other necessary command line switches

- -O2 -fdump-tree-all
- -O3 enables -ftree-vectorize. Other flags must be enabled explicitly

Other useful options

- Suffixing -all to all dump switches
- -S to stop the compilation with assembly generation
- -fverbose-asm to see more detailed assembly dump

SYNTAX ANALYSIS (PARSING)

The journey in this phase of a compiler as it is stated in the syllabus.

Module II (From the Syllabus) :

Introduction to Syntax Analysis, Elimination of Ambiguity, Left Recursion and Left Factoring, Recursive and Non-Recursive Top-Down Parsers, Bottom-up Parsers: Shift Reduce Parser techniques and conflicts, all variants of LR Parsers, Handling Ambiguous grammar in Bottom- Up Parsing, Error handling while parsing, the Parser generator YACC. **(15 Lectures)**

SYNTAX ANALYSIS OR PARSING

1.Motivation

After lexical analysis, syntax analysis (or parsing) is the next phase in compiler design. Syntax analyser (or parser) is the name of that part of a compiler which performs this task.

```
main ()
{
int i,sum;
sum = 0;
for (i=1; i<=10; i++)
sum = sum + i;
printf("%d\n",sum);
}
```

main	()	{	int	i	,	sum	;	sum	=	0	;	for	(
i	=	1	;	i	<=	10	;	i	++)	;	sum	=	sum	+	i
;	printf	("%d\n"	,	sum)	;	}								

In order to check the syntactic validity of the code fragment, we need the rules for constructing the various linguistic features used there.

The given code is a definition of function named `main()`. The body of the function has declarative statement, a loop construct, an assignment statement, a function call for output, etc. Let us use our intuition and exposure to C to detail out a specification of some of these constructs in a semi formal manner.

- Function definition usually requires a `return_type` for the return value of a function if any, its name, zero or more arguments enclosed in a pair of parentheses, its body of statements enclosed in another pair of curly braces, and so on. In a symbolic manner, we shall write rules of the form :

$$\text{language_feature} \rightarrow \text{description}$$

Choosing names to describe a feature in terms of its sub features is a possible approach.

- $\text{fundef} \rightarrow \text{rettype fname (arglist) \{body\}}$

The name `fundef` is chosen to define the structure of a function definition in C. The terms used on the rhs of \rightarrow describe the components that form a definition. The terms `rettype`, `fname`, `arglist`, `body` are the names chosen to describe the return-type, function name, list of parameters followed by the statements that usually constitute the body of a function.

- $\text{rettype} \rightarrow \text{int} \mid \text{void} \mid \epsilon$

A few common return types are specified for the present discussion. In general one would have to include all the valid return types permitted in C. The symbol ϵ denotes an empty string indicating return type is not mandatory and may be skipped.

- $\text{fname} \rightarrow \text{id}$

A function name is specified as any valid identifier (`id`) in C.

- The two parentheses, ‘(’ and ‘)’ are mandatory part of a function definition.
- The name `arglist` specifies the manner in which parameters are specified in C, however since our `main()` does not use them, we shall use ϵ for this feature.

$$\text{arglist} \rightarrow \epsilon$$

A general description of `arglist` would be of the form

$$\begin{aligned} \text{arglist} &\rightarrow \text{arglist} \mid \epsilon \\ \text{arg} &\rightarrow \text{type id} \mid \text{arg, type id} \end{aligned}$$

- $\text{body} \rightarrow \text{decl slist} \mid \epsilon$

decl denotes a declaration statement while slist stands for a list of statements.

- $\text{decl} \rightarrow \text{type idlist}; \quad \text{idlist} \rightarrow \text{idlist, id} \mid \text{id} \quad \text{type} \rightarrow \text{int} \mid \text{float}$

The descriptions given above are adequate to generate a single declaration statement with a type followed by a list of comma separated identifiers.

Let us consolidate all the semi-formal specifications written above as follows. These are to be taken as indicative and complete specifications are significantly larger and involve more features even to specify the simple subset of C chosen here.

```

fundef  $\rightarrow$  rettype fname (arglist) {body}
rettype  $\rightarrow$  int  $\mid$  void  $\mid$   $\epsilon$ 
fname  $\rightarrow$  id
arglist  $\rightarrow$   $\epsilon$ 
body  $\rightarrow$  decl slist  $\mid$   $\epsilon$ 
decl  $\rightarrow$  type idlist;
idlist  $\rightarrow$  idlist, id  $\mid$  id
type  $\rightarrow$  int  $\mid$  float

```

The objective is to get some insight into the process of syntax checking of a code fragment. We shall use the earlier code fragment along with the specification of constructs listed above.

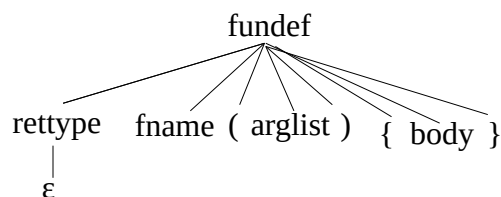
Objective : To perform a parsing or syntax analysis of the following code using the specifications given above. Let us assume that we are told that a function definition is to be analysed.

```
main () {int i, sum; sum = 0; for (i=1; i<=10; i++) sum = sum + i; printf("%d\n", sum); }
```

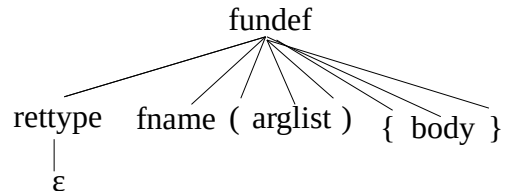
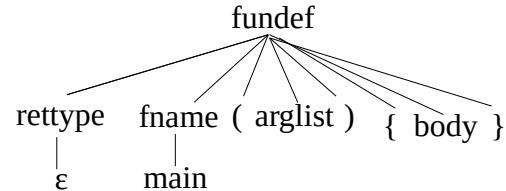
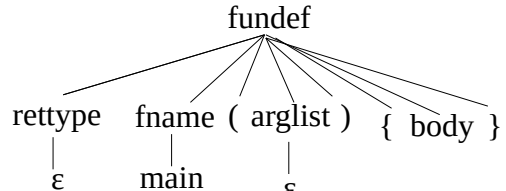
The current token is : main

To match with a function definition, we use the first description.

The specification $\text{fundef} \rightarrow \text{rettype fname (arglist) \{body\}}$ requires that the 8 components that it comprises of needs to be matched.



We shall use this tree structure to go ahead with intuitive parsing of the input code fragment. The tree is placed in the last column of a table that shows the progress in parsing along with the associated actions that are taken.

Current token	Action Taken	Checking for Syntactic Structure
main	Since there is no return type, we use the descriptions $rettype \rightarrow \epsilon$, and proceed.	 <pre> graph TD fundef --> rettype fundef --> fname fundef --> LP["("] fundef --> arglist fundef --> RP[")"] fundef --> LBrace["{"] fundef --> body fundef --> RBrace["}"] rettype --> epsilon["ε"] </pre>
main	main is an identifier, which matches with fname and the tree is extended accordingly.	 <pre> graph TD fundef --> rettype fundef --> fname fundef --> LP["("] fundef --> arglist fundef --> RP[")"] fundef --> LBrace["{"] fundef --> body fundef --> RBrace["}"] rettype --> epsilon["ε"] fname --> main </pre>
(matches the structure	Same tree. Now arglist is required
)	No arguments given in the input; use the description $arglist \rightarrow \epsilon$ The input has matched till arglist of the specification of a function definition	 <pre> graph TD fundef --> rettype fundef --> fname fundef --> LP["("] fundef --> arglist fundef --> RP[")"] fundef --> LBrace["{"] fundef --> body fundef --> RBrace["}"] rettype --> epsilon1["ε"] fname --> main arglist --> epsilon2["ε"] </pre>
)	Matches with the specification. Get the next token.	Same Tree.
{	Matches with specification. Get the next token.	Same Tree. Now the structure of body is expected to be seen in the input.
int i, sum;	Proceeding along the same lines, what would the tree structure after ';' has been read from the input	<p>The tree is drawn below.</p> <p>The nonterminal slist remains to be explored with the remaining input :</p> <pre>sum = 0;for (i=1; i<=10; i++)sum = sum + i;printf("%d\n",sum);}</pre>

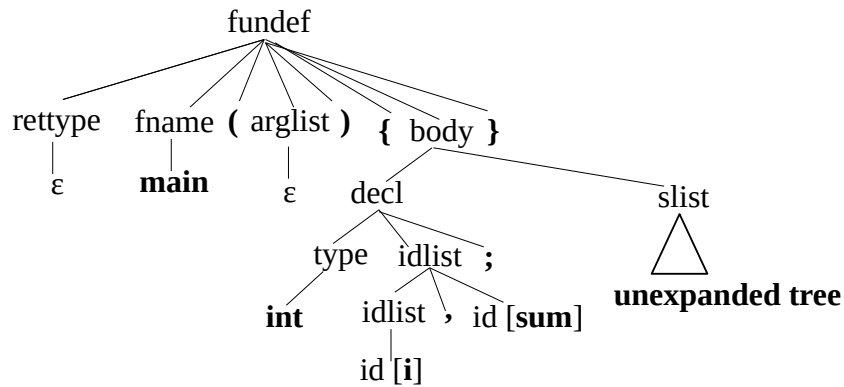


Figure : Tree after partially processing the input : `main () {int i, sum; sum = 0;}`

The preceding material introduces the essential issues in parsing while skipping the details. The terms and basic concepts of parsing are addressed in the rest of the document.

Basic Issues in Parsing

Q. What does a parser (or syntax analyzer) do?

- groups tokens appearing in the input and attempts to identify larger structures in the program. This amounts to performing a syntax check of the program.
- makes explicit the hierarchical structure of the token stream. Associated is the issue of representation - how should the syntactic structure of a program be explicitly captured ? This information is normally required by subsequent phases.

Q. How to specify Programming Language (PL)Syntax ?

It is obvious that a parser must be provided with a description of PL syntax. How to describe the same is an important question and the related issues are :

1. The specification be precise and unambiguous.
2. The specification be complete, that is cover all the syntactic details of the language.
3. Specification be such that it be a convenient vehicle for both the language designer and the implementer.

We have already studied a formalism called Context Free Grammar in the course on “Formal Languages and Automata”. In this module we shall examine the extent to which CFG meets the requirements stated above in order to apply it to parsing.

Q. How to represent the input after it has been parsed ?

We shall study one representation known as parse tree in this context. The same representation has many variants, such as Abstract Syntax Tree (AST) and others.

Q. What are the parsing algorithms, how they work and what are their strengths / limitations ?

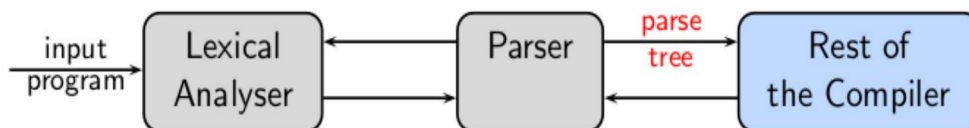
This is the primary concern of the module. We shall discuss two different approaches to parsing , known as, top-down parsing and bottom- up parsing, and study some parsing algorithms of both types. The two parsers are not equivalent in their applicability and the strengths of parser in the family of parsers shall be discussed here.

- In syntax analysis phase, we are not interested in determining what the program does (known as the semantics of the program) and hence such issues are not addressed here. The semantic analysis aspects are discussed separately.

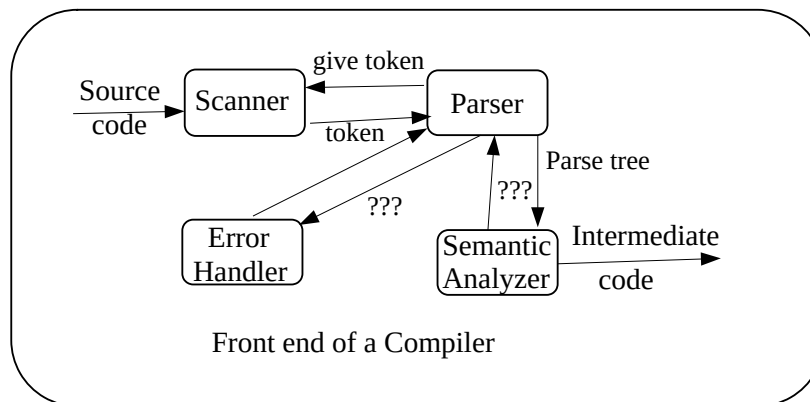
Q. How are parsers constructed ?

- Till about 2-3 decades back, parsers (in fact the entire compiler) was written manually.
- A better understanding of the parsing algorithms has led to the development of special tools which can automatically generate a parser for a given PL.
- As we shall see later both top-down and bottom-up parsers can be automatically generated.
- A compiler generation tool, called YACC (Yet Another Compiler Compiler), generates a bottom-up parser and is available on Unix. Another such tool is BISON available in the GNU public domain software.
- Similarly ANTLR (ANother Tool for Language Recognition), is a parser generator tool that generates top down parsers. However top down parser generators are outside the scope of this course.

Interaction of Parser with scanner and the rest of the compiler modules was shown earlier as depicted in the following figure.



Let us extend the earlier figure to include the next phase of the compiler known as semantic analyzer.



The legend “???” indicates that the exact interface between the modules depends on the concerned modules. It should be noted that though syntax and semantic analysis are two distinct phases of a compiler because they perform different functions, that are often integrated into the same pass of the compiler in practice.

Specification of PL Syntax

Program in any language is essentially a string of characters from its alphabet; however an arbitrary string is not necessarily a valid program.

- The problem of specifying syntax is to determine those strings of characters that represent valid programs (programs that are syntactically correct). Rules which identify such valid programs spell out the syntax of the language.
- Recall that tokens are the lexical structures of a language and can be defined precisely and formally. This helps in the automatic generation of lexical analyzers.

What is the situation with syntactic structures and parsing ?

- Syntactic structures are grouping of tokens. The language definition provides the syntactic structures that are permitted. A few syntactic structures common across PLs are

expression	declaration	statement	block
compound-statement	function	program	

- Syntax of expressions is nontrivial - depends on the richness and the properties of the operators supported in the language. The operator set of C / C++, that has been shared with you, is a glaring example of this fact.

- Since you are quite familiar with the syntax of C, let us use a different PL known as Pascal to informally describe a possible syntax for **variable declarations** in Pascal (was very popular in the 70s but not used much today)

- a stream of tokens that has keyword **var** as the first token
- followed by one or more tokens of type **identifier**, separated by token **comma**
- followed by token **colon**
- followed by any one of the tokens, **integer** or **real**
- followed by token **semicolon**

An example of a variable declaration in Pascal is : **var a,b,c : integer;**

Contrast this to an equivalent declaration in C : **int a, b, c;**

Even for such a simple syntactic structure, the problems with informal description (the second rule specially) are evident. This is the motivation for formal specification of syntax.

To summarize, the following questions address some of the basic issues related to specification of syntax that a parser writer must be aware of.

A few relevant questions in this context.

Q1. Does there exist formalism that can be used to describe all the syntactic details of a PL ? If the answer is yes, then can the same formalism be used to write a parser for the language ?

Q2. If one uses a formalism that captures most of the syntactic details (but not all) of a PL and uses the same to write a parser, then when and how are the missing features taken care of ?

Q3. Given a formal mechanism , can the syntax of a language be specified uniquely ?

Q4. If the answer to above is no, then which among the several specifications should be preferred while constructing parsers and why ?

Q5. Is it useful for a compiler writer to know how to write syntactic specification that is suitable for parsing ?

Basic Concepts in Parsing : Context Free Grammar

We introduce a notation called Context Free Grammar (CFG) informally and then give a formal definition.

- It is a notation for specifying programming language syntax.
- Identify the syntactic constructs of the language , such as expression, statement, etc. and express its syntax by means of rewriting rules. We continue with sample specifications of a declaration statement in Pascal and C, as given below. The

Grammar for Single Declaration in Pascal	Grammar for Single Declaration in C
declaration \rightarrow var decl-list decl-list \rightarrow list : type; list \rightarrow id list , id type \rightarrow integer real	declaration \rightarrow type decl-list ; decl-list \rightarrow id list , id type \rightarrow int float

The symbols that appear in an input are marked in bold in the table above. Concatenation is the assumed operator between the symbols in the right hand side (rhs).

- Apart from tokens (also called as **terminals**), other symbols have been used in the **rewrite rules** above. These symbols called **nonterminals** (or syntactic category) do not exist in a source program.
- A nonterminal symbol only is allowed to appear in the left hand side (lhs) of a rewrite rule. However in the rhs, both nonterminal and terminal symbols may appear.
- The rhs of a rewrite rule (also called a **production rule**) specifies the syntax of the lhs nonterminal. It may contain both kinds of symbols.
- Since the issue is to parse (and compile) programs, a nonterminal that specifies the syntax for a program must be present in the CFG. For instance a C program may be defined as a collection of one or more function definitions preceded by zero or more global declarations. The nonterminal program is called the **start symbol**. The nonterminals in the following are marked in *italics*, separate them from the terminal symbols.

program \rightarrow *global-decls* *func-list*
func-list \rightarrow *func* | *func func-list*
func \rightarrow *ret-type func-name (arglist) { func-body }*
...

Example of the structure of a program in pascal :

program \rightarrow **program id** (*list*) ; *decls* *compound-statement*

- Other nonterminals are introduced to specify the various parts of the start symbol, such as **program** above, in a structured manner.

A CFG is formally defined in the following . It has four components, G is the grammar, and written as $G = (T, N, S, P)$, where

- T is a finite set of terminals (or tokens)
- N is a finite set of nonterminals
- S is a special nonterminal (from N) called the start symbol
- P is a finite set of production rules of the form

$$A \rightarrow \alpha, \text{ where } A \text{ is from } N \text{ and } \alpha \text{ is from } (N \cup T)^*$$

There can be more than one definition for a nonterminal (different rhs for the same lhs) in which case the definitions are separated by the symbol ‘|’.

Q. Why the term context free ?

1. $\text{left_symbol} \rightarrow \text{right_symbols}$ is the only kind of rule permitted in a CFG; where left_symbol is a single nonterminal and right_symbols is a string from $(N \cup T)^*$
2. Rules are used to replace an occurrence of the lhs nonterminal by its rhs. In a CFG, the replacement is made regardless of the context of the lhs nonterminal (symbols surrounding it).

Grammars, as we know from our experience with languages (natural or PL), are used to define languages.

Q. What is the relation between CFG of a PL and the PL itself ?

1. Intuitively, a PL may be defined by the collection of all valid programs that can be written in that language.
2. A grammar is used to define a language in the sense that it provides a means of generating all its valid programs.
3. Beginning with the start symbol of the grammar and using production rules repeatedly (for replacing nonterminal symbols), one can produce a string comprising terminals only. Such sequence of replacements is called a derivation.
4. The set of all possible terminal strings (elements of T^*) that can be derived from the start symbol of a CFG G is an important collection. Informally this set of strings is the language denoted by G .

Example : Consider a CFG , $G = (T, N, S, P)$ with $N = \{list\}$, $S = list$, $T = \{id, , \}$ and P containing 2 rules, $P :$

1. $list \rightarrow list, id$
2. $list \rightarrow id$

which could also be combined to the compact form $list \rightarrow list, id \mid id$

A derivation is traced out as follows

list	derives	list, id
	derives	list, id
	derives	list, id, id
	derives	id, id, id

The derivation process stops when the string “**id , id , id**” is reached because this string does not any nonterminals. Using the derivation process, we can see that grammar G is capable of generating terminal strings such as

$L(G) = \{id\ id, id\ id, id, id\ id\ \dots\}$.

In English, we may describe the language denoted by G , that is $L(G)$ as the non-empty set of strings that comprise of one or more **id** separated by commas (,).

Notational Conventions

Since we have to frequently refer to terminals and nonterminals, the following conventions are commonly used in the literature on CFG and Parsers.

Symbol type	Convention
single terminal	small case letters from the start of the English Alphabet, such as {a, b, c,...}, operators, delimiters, keywords, etc.
single nonterminal	upper case letters from the start of the English Alphabet, such as {A, B, C,...}, names such as <i>declaration</i> , <i>list</i> , <i>expression</i> , etc. <i>S</i> is typically reserved for the start symbol.
single grammar symbol	upper case letters at the end of the English Alphabet, such as {X, Y, Z,...}
string of terminals	small case letters at the end of the English Alphabet, such as {x, y, z,...}
string of grammar symbols	Greek letters such as { α , β , γ , δ , ...}
null string	ϵ

Formal Definitions

The concepts and terms that have been introduced informally so far, are now formally defined.

Let $A \rightarrow \gamma$ be a production rule.

Consider a string $\alpha A \beta$ from $(N \cup T)^*$. Using the convention stated above, the string $\alpha A \beta$ indicates that there is a nonterminal A in the string which is surrounded on either side by some grammar symbols. The key idea for choosing such a string is to notify the presence of a nonterminal A in it.

1. Replacing the nonterminal A , by its definition $A \rightarrow \gamma$ in the string $\alpha A \beta$ above, yields the new string $\alpha \gamma \beta$. Formally this is stated as $\alpha A \beta$ derives $\alpha \gamma \beta$ in a derivation of one step. A concise form of writing a **one step derivation** is :

$\alpha A \beta \Rightarrow \alpha \gamma \beta$, where the symbol \Rightarrow stands for **derives in one step**.

2. If $\alpha_1, \alpha_2, \dots, \alpha_n$ are arbitrary strings of grammar symbols, such that $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow \dots \Rightarrow \alpha_n$, then we say that α_1 derives α_n .

3. If a derivation comprises of zero or more steps, the symbol \Rightarrow^* is used. Clearly for any string α , it is obvious that $\alpha \Rightarrow^* \alpha$ is true.

4. If a derivation comprises of one or more steps, the symbol \Rightarrow^+ is used.

5. It is interesting to note that $\Rightarrow, \Rightarrow^*$ and \Rightarrow^+ are relations over $(N \cup T)^*$ in the set theoretic sense of relation; where \Rightarrow^* is the reflexive transitive closure of \Rightarrow , while \Rightarrow^+ is the transitive closure of \Rightarrow . The last of these three relations is used to define the concept of a language.

6. The language $L(G)$ denoted by a context free grammar G is defined by $L(G) = \{ w \mid S \Rightarrow^+ w, w \in T^* \}$. Such a string w is called a **sentence** in $L(G)$.

7. A string $\alpha, \alpha \in (N \cup T)^*$, such that $S \Rightarrow^* \alpha$, is called as a sentential form in $L(G)$.

8. Grammars $G_1 = (T, N_1, S_1, P_1)$ and Grammars $G_2 = (T, N_2, S_2, P_2)$ are said to be equivalent, written as $G_1 \equiv G_2$, if they generate the same language, i. e., $L(G_1) = L(G_2)$.

EXAMPLE 1: Illustrations of Derivations and sentential forms

Consider the 3 grammars given below.

$G_1 = (T, N_1, L, P_1)$, where $T = \{ , id \}$; $N_1 = \{ L \}$ and $P_1 = \{ L \rightarrow L , id \mid id \}$.

$G_2 = (T, N_2, L, P_2)$, where $N_2 = \{ L \}$ and $P_2 = \{ L \rightarrow id , L \mid id \}$

$G_3 = (T, N_3, L, P_1)$, where $N_3 = \{ L, L' \}$ and P_3 has 3 rules given below.

$$\begin{aligned} L &\rightarrow id L' \\ L' &\rightarrow , id L' \mid \epsilon \end{aligned}$$

It can be shown that the set of strings generated by all the grammars is $\{id \ id, id \ id, id, id \ \dots\dots\dots\}$.
Therefore since $L(G_1) = L(G_2) = L(G_3)$, all the three grammars are equivalent, or $G_1 \equiv G_2 \equiv G_3$.

Consider grammar G_1 and construct a derivation of the string “id , id, id” from the start symbol L .

$$L \Rightarrow L , id \Rightarrow L , id , id \Rightarrow id , id , id$$

The derivation involves 4 sentential forms $\{L \ L , id \ L, id, id \ id, id, id\}$ in the order they are generated of which the last one is also a sentence.

What is the type of this derivation among {leftmost, rightmost, arbitrary}?

The derivation of the string “id , id, id” using grammar G_2 turns out to be the following.

$$L \Rightarrow id , L \Rightarrow id , id , L \Rightarrow id , id , id$$

The derivation above involves 4 sentential forms $\{L \ id, L \ id, id, L \ id, id, id\}$ in the order they are generated with the last sentential form also being a sentence.

What is the type of this derivation among {leftmost, rightmost, arbitrary}?

The derivation of the string “id , id, id” using grammar G_3 is given by the following.

$$L \Rightarrow id L' \Rightarrow id , id L' \Rightarrow id , id , id L' \Rightarrow id , id , id$$

This derivation involves 5 sentential forms $\{L \ id L' \ id, id L' \ id, id, id L' \ id, id, id\}$ in the order generated with the last sentential form also being a sentence. What is the type of this derivation among {leftmost, rightmost, arbitrary}?

Derivations And Their Use

1. A derivation of w from S is a proof that w is in the language of G , or $L \in L(G)$.
2. Derivation provides a means for generating the sentences of $L(G)$.
3. For constructing a derivation from S , there are options at two levels
 - choice of a nonterminal to be replaced among several others
 - choice of which particular rule (since there could be more than one rules for a given nonterminal) corresponding to the nonterminal selected.
4. Instead of choosing the nonterminal to be replaced during derivation, in a given sentential form, in an arbitrary fashion, it is possible to make an uniform choice at each step. Two natural selections for the replacement of a nonterminal are
 - replace the leftmost nonterminal in a sentential form
 - replace the rightmost nonterminal in a sentential form

The corresponding derivations are known as leftmost and rightmost derivations respectively.

5. Given a sentence w of a grammar G , there may be several distinct derivations for w .

EXAMPLE 2 : Consider the following expression grammar, the name of the grammar is pertinent, since it generates arithmetic expressions involving 5 terminal symbols $\{ \text{id } () + * \}$. We shall write a CFG by specifying the production rules henceforth, since the other components can be inferred from the context.

CFG for expressions	Using the Grammar for Derivations
$E \rightarrow E + T \mid T$	The grammar has 6 production rules. The start symbol is nonterminal E . There are 3 nonterminals $\{E, T, F\}$. There are 5 terminals $\{ \text{id } () + * \}$
$T \rightarrow T * F \mid F$	
$F \rightarrow (E) \mid \text{id}$	

- Let us derive the sentence **id** using the production rules. We shall mark the nonterminal, that is chosen for replacement, by double underline in the sentential form. It may be noted the derivation of **id** from E took 3 steps and there was no choice for selection of a nonterminal as each sentential form had a single nonterminal.

$$\underline{\underline{E}} \Rightarrow \underline{\underline{T}} \Rightarrow \underline{\underline{F}} \Rightarrow \text{id}$$

However, there was a choice for selecting one of the two alternates for each of E, T and F. For example, one could have the other alternative rule for E in the first step resulting in

$$\underline{\underline{E}} \Rightarrow E + T$$

Unfortunately from the sentential form “E + T”, it is not possible to derive **id**, regardless of which nonterminal was chosen for replacement.

- Let us now derive the sentence “**id + id * id**”

$$\begin{aligned} \underline{\underline{E}} &\Rightarrow E + \underline{\underline{T}} \Rightarrow E + \underline{\underline{T}} * F \Rightarrow E + \underline{\underline{F}} * F \Rightarrow \underline{\underline{E}} + id * F \Rightarrow \underline{\underline{T}} + id * F \\ &\Rightarrow \underline{\underline{F}} + id * F \Rightarrow id + id * \underline{\underline{F}} \Rightarrow id + id * id \end{aligned}$$

In the derivation above, the nonterminals chosen for expansion in the 2nd sentential form onwards are {last, middle, middle, first, first, first, last} and do not have any fixed pattern. Such a derivation will be referred to as an arbitrary derivation.

- Consider the following derivation of the same sentence “**id + id * id**”

$$\begin{aligned} \underline{\underline{E}} &\Rightarrow E + \underline{\underline{T}} \Rightarrow E + T * \underline{\underline{F}} \Rightarrow E + \underline{\underline{T}} * id \Rightarrow E + \underline{\underline{F}} * id \Rightarrow \underline{\underline{E}} + id * id \\ &\Rightarrow \underline{\underline{T}} + id * id \Rightarrow \underline{\underline{F}} + id * id \Rightarrow id + id * id \end{aligned}$$

In the derivation above, the nonterminals chosen for expansion is always the last nonterminal (or the right-most nonterminal) in all the sentential forms. Such a derivation is referred to as a rightmost derivation.

- A leftmost derivation of the same sentence will be the one in which the leftmost nonterminal is selected for replacement in every sentential form. The leftmost derivation is shown below.

$$\begin{aligned} \underline{\underline{E}} &\Rightarrow \underline{\underline{E}} + T \Rightarrow \underline{\underline{T}} + T \Rightarrow \underline{\underline{F}} + T \Rightarrow id + \underline{\underline{T}} \Rightarrow id + T * \underline{\underline{F}} \\ &\Rightarrow id + \underline{\underline{T}} * id \Rightarrow id + \underline{\underline{F}} * id \Rightarrow id + id * id \end{aligned}$$

The two special derivations are usually marked by adding the label **rm** or **lm** to the derivation symbol \Rightarrow , depending on whether the derivation is rightmost or leftmost. The derivations are rewritten below with the labels as stated above.

$$\begin{aligned} \underline{\underline{E}} &\Rightarrow_{rm} E + \underline{\underline{T}} \Rightarrow_{rm} E + T * \underline{\underline{F}} \Rightarrow_{rm} E + \underline{\underline{T}} * id \Rightarrow_{rm} E + \underline{\underline{F}} * id \Rightarrow_{rm} \underline{\underline{E}} + id * id \\ &\Rightarrow_{rm} \underline{\underline{T}} + id * id \Rightarrow_{rm} \underline{\underline{F}} + id * id \Rightarrow_{rm} id + id * id \\ \underline{\underline{E}} &\Rightarrow_{lm} \underline{\underline{E}} + T \Rightarrow_{lm} \underline{\underline{T}} + T \Rightarrow_{lm} \underline{\underline{F}} + T \Rightarrow_{lm} id + \underline{\underline{T}} \Rightarrow_{lm} id + T * \underline{\underline{F}} \\ &\Rightarrow_{lm} id + \underline{\underline{T}} * id \Rightarrow_{lm} id + \underline{\underline{F}} * id \Rightarrow_{lm} id + id * id \end{aligned}$$

Derivations And Parse Trees

There is an equivalent form of depicting a derivation that is pictorial in nature and is called a parse tree. A parse tree for a context free grammar, is a tree having the following properties :

1. root of the tree is labeled with the start symbol S ,
2. each leaf node is labeled by a token or by ϵ ,
3. an internal node of the tree is labeled by a nonterminal,
4. if an internal node has A as its label, then the children of this node from left to right are labeled with $X_1, X_2, X_3, \dots, X_n$ when there is a production of the rule
$$A \rightarrow X_1 X_2 X_3 \dots X_n$$
where X_i is a grammar symbol.
5. the leaves of the tree read from left to right give the yield of the tree; essentially the sentence generated or derived from the root.

An example of a parse tree is given later.

Q. How to construct a parse tree from a derivation ?

Let $S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n = z$ be a derivation of sentence z from S in derivation steps.

The corresponding parse tree may be constructed as follows

- create a root labeled with S
- for each sentential form, α_i , $i \geq 1$, construct a parse tree with the yield of α_i
- If the tree for α_{i-1} is available, the tree for α_i is easily constructed (by using induction), as given below.

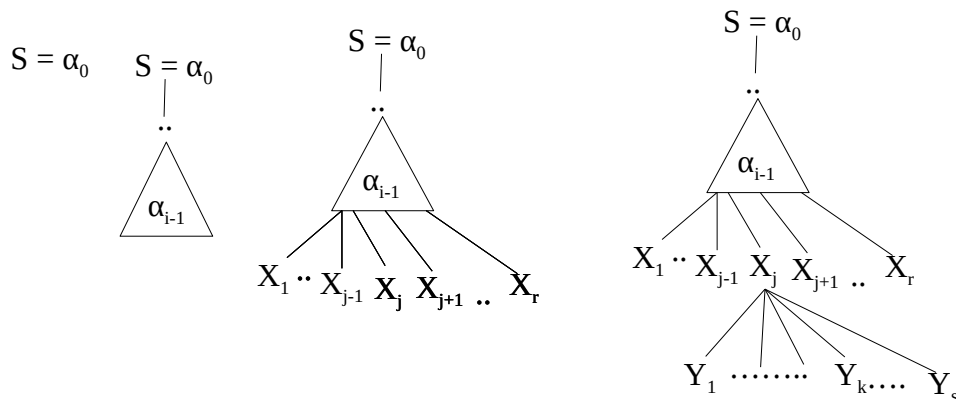
Let $\alpha_{i-1} = X_1 X_2 X_3 \dots X_r$

$\Rightarrow X_1 X_2 X_3 \dots X_{j-1} \beta X_{j+1} \dots X_r = \alpha_i$

where $X_j \rightarrow \beta$ is the rule used and let β be $Y_1 Y_2 \dots Y_s$

- The node (leaf) corresponding to X_j in the parse tree is expanded by
 - creating s number of children for the node corresponding to X_j , and
 - labeling these children with $Y_1 Y_2 \dots Y_s$ in the order from left to right.

The creation is progressively shown in the following figure. The first node is the root of the tree. The 2nd tree assumes that the tree has been drawn till the sentential form $S = \alpha_0 \Rightarrow^* \alpha_{i-1}$; the 3rd tree shows tree rooted at α_{i-1} ; with its r children labeled as X_i , $1 \leq i \leq r$ and the final tree shows the situation corresponding to the sentential form : $S = \alpha_0 \Rightarrow^* \alpha_i$



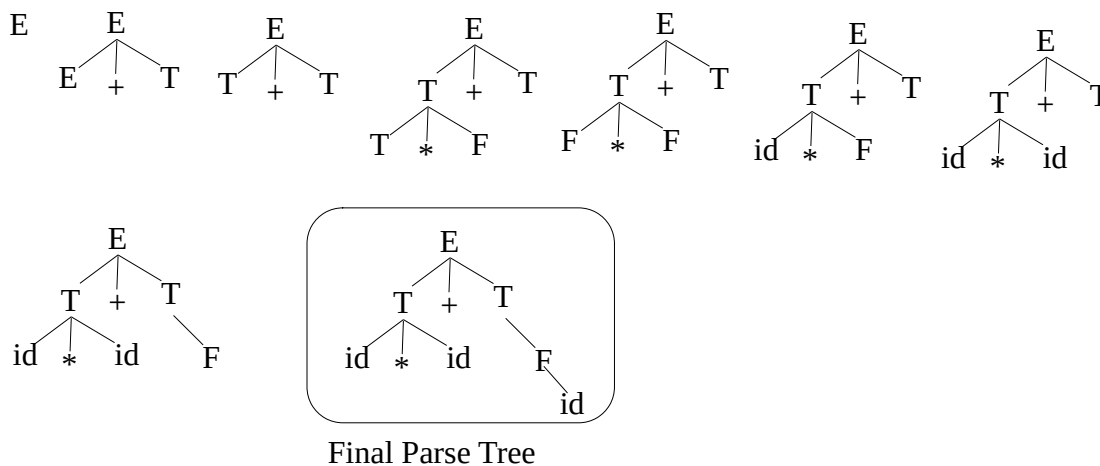
EXAMPLE : Consider the following expression grammar.

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

Leftmost derivation of $\text{id} * \text{id} + \text{id}$

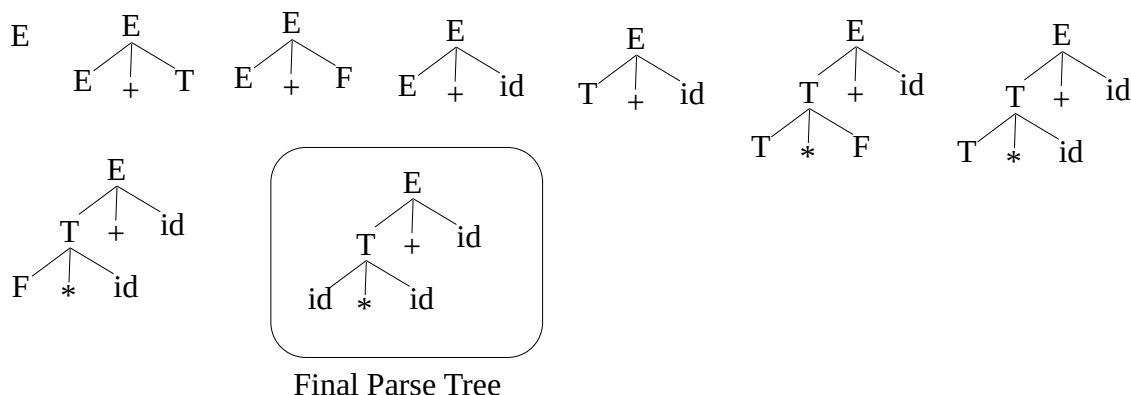
$$\begin{aligned} \underline{E} &\Rightarrow_{\text{lm}} \underline{E} + T \Rightarrow_{\text{lm}} \underline{T} + T \Rightarrow_{\text{lm}} \underline{T} * F + T \Rightarrow_{\text{lm}} \underline{F} * F + T \Rightarrow_{\text{lm}} \text{id} * \underline{F} + T \\ &\Rightarrow_{\text{lm}} \text{id} * \text{id} + \underline{T} \Rightarrow_{\text{lm}} \text{id} * \text{id} + \underline{F} \Rightarrow \text{id} * \text{id} + \text{id} \end{aligned}$$

The parse trees as the derivation proceeds are shown below.



Rightmost derivation of $\text{id} * \text{id} + \text{id}$

$$\begin{aligned} \underline{E} &\Rightarrow_{\text{rm}} E + \underline{T} \Rightarrow_{\text{rm}} E + \underline{F} \Rightarrow_{\text{rm}} \underline{E} + \text{id} \Rightarrow_{\text{rm}} \underline{T} + \text{id} \\ &\Rightarrow_{\text{rm}} T * \underline{F} + \text{id} \Rightarrow_{\text{rm}} \underline{T} * \text{id} + \text{id} \Rightarrow_{\text{rm}} \underline{F} * \text{id} + \text{id} \Rightarrow_{\text{rm}} \text{id} * \text{id} + \text{id} \end{aligned}$$



The parse tree corresponding to either of the two derivations produces the same parse tree. The tree is given below. The tree shows that its independent of the derivation that produces it, however this fact holds only when there exists a unique derivation of the sentence (both leftmost and rightmost).

Derivations And Parse Trees

The following summarize some interesting relations between the two concepts.

1. Parse tree filters out the choice of replacements made in the sentential forms.
2. Given a derivation for a sentence, one can construct a parse tree for the sentence. In fact one can draw the parse tree as the derivation proceed.
3. Even while several distinct derivations may exist for a given sentence, they usually correspond to a single parse tree.
4. Given a parse tree for a sentence, it is possible to construct a unique leftmost and a unique rightmost derivation.

Q. Can a sentence have more than one distinct parse trees corresponding to it ? Construct examples, if such is possible.

Ambiguous Grammar

A context free grammar G that produces more than one parse tree for a sentence of $L(G)$ is defined to be an ambiguous grammar.

Equivalently a context free grammar that has two or more leftmost (or rightmost) derivations for a sentence is an ambiguous grammar.

Ambiguous grammars are usually not suitable for parsing, because of the following reasons.

1. A parse tree would be used subsequently for semantic analysis; more than one parse tree would imply several interpretations and there is no obvious way of preferring one over another.
2. How does one detect that a given context free grammar is indeed ambiguous ?
3. Since multiple parse trees, even for a single sentence, renders the grammar ambiguous, is an algorithmic solution feasible ?
4. What can be done with ambiguous grammars in the context of parsing?
 - Rewrite the grammar such that it becomes unambiguous.
 - Use the grammar but supply disambiguating rules (used by the tool YACC).

Writing A CFG For PL Features

Given a PL feature, one may write several distinct but equivalent context free grammars for describing its syntax.

There are several thumb rules that can be meaningfully used in writing a grammar that is better suited for parsing.

- Many syntactic features of a language are expressed using recursive rules. Usually these can be written in either left recursive or right recursive form. While both forms may be equivalent in expressiveness, they have different implications in parsing. As an example, consider syntax for declarations :

Using a left recursive rule (the first symbol in the rhs of a rule is the same nonterminal as that in the lhs) such as the rule given below.

```
D → var list : type ;  
type → integer | real  
list → list , id | id
```

Using a right recursive rule (the last symbol in the rhs of a rule is the same nonterminal as that in the lhs) such as

```
D → var list : type ;  
type → integer | real  
list → id , list | id
```

Both the grammars are equivalent in terms of expressiveness, the underlying languages which can contain potentially infinite number of id, are same for both grammars.2. Writing a grammar for expressions; points of concern here are the operators and their properties.

Associative property: This property is useful to answer the question, “ In the absence of parenthesis, how an expression such as $a \theta b \theta c$, is to be evaluated” ? Here θ is some binary operation.

- An operator is classified as either **left** or **right** associative, depending upon whether an operand with same θ on both its sides is to be consumed by the θ placed to its left or right. Operators $+$, $-$, $*$ and $/$ are left associative while \uparrow (exponentiation in Pascal) or the assignment operator '=' in C are right associative.
- To honor the left(or right) associative properties of operators, left (or right) recursive rules are useful.

Precedence Property : Several operators have different precedence attached to them and an expression containing them has to be evaluated in accordance to the precedence values, else different interpretations may arise. Programming Language definition specifies the relative precedence of its permissible operators.

- Typically the operators such as $\{ *, / \}$ have higher precedence than $\{ +, - \}$; in the presence of these operators with varying precedence values, grammar rules for writing the syntax of expressions that involve these 2 sets of operators is constructed as follows.
 - A nonterminal for each of the two sets are chosen, say *term* and *exp* respectively.
 - The rules for the operators with lower precedence are :
$$\text{exp} \rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term}$$

Notice the use of left recursion and the use of 2 nonterminals.
 - To write the rules for the other operator class, $\{ *, / \}$ the basic units in expressions are needed. Another nonterminal, say, *factor*, is chosen for the purpose.
$$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}$$

- Finally, The rules for the basic unit are written :
factor \rightarrow id | (exp)
- Provide intuitive reasoning as to why the above empirical rules are adequate.

Disambiguating a Grammar : There are several useful transformations that make a CFG more amenable for parsing. Disambiguating a grammar is one of them.

- The following grammar for expressions is ambiguous, but very concise. Note that there are two instance of operator $-$, one is unary minus and the other one is a binary minus.

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid \text{id}$$

- Prove the fact that the CFG given above is ambiguous
- Using the empirical rules stated in item 2 above, and along with the properties of all the involved operators, we know how to write an equivalent CFG that is unambiguous.
- Consider another grammar given below. This grammar is supposed to generate nested if-then-else statements in PLs. A generic form of this conditional statement is given here, languages may have a small variation in the syntax for this construct.

$$S \rightarrow \text{if } \mathbf{c} \text{ then } S \\ \mid \text{if } \mathbf{c} \text{ then } S \text{ else } S \mid \text{other}$$

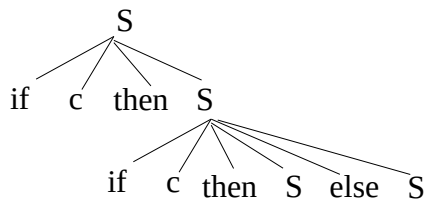
The keywords are marked in bold, the other symbols are nonterminals. We shall treat c as a terminal in this context since the focus is on conditionals. We can ignore the nonterminal, *other*, because that is supposed to take care of statements that are not conditionals.

- Is the grammar above ambiguous? Consider the sentence : “if c then if c then S else S ” and parse this sentence using the grammar for conditionals.

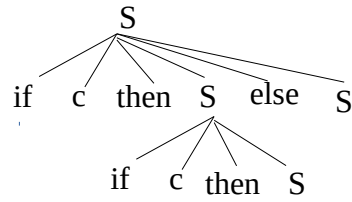
$$S \Rightarrow \text{if } \mathbf{c} \text{ then } S \Rightarrow \text{if } \mathbf{c} \text{ then if } \mathbf{c} \text{ then } S \text{ else } S \quad (1)$$

$$S \Rightarrow \text{if } \mathbf{c} \text{ then } \underline{S} \text{ else } S \Rightarrow \text{if } \mathbf{c} \text{ then if } \mathbf{c} \text{ then } S \text{ else } S \quad (2)$$

Two distinct leftmost derivations are seen above, whereby the the grammar given above is proved to be ambiguous. The same observation can also be observed if parse trees are drawn for these two derivations. The two parse trees are very different showing clearly the ambiguity.



Parse Tree for (1)



Parse Tree for (2)

Q. Is it possible to write an unambiguous grammar for conditional statements ?

In order to find an answer to the question, it is insightful to examine a sentence comprising of nested conditional statements and determine how PLs decipher the same.

Let us examine the following input for the purpose :

if then if then if then else if then else else

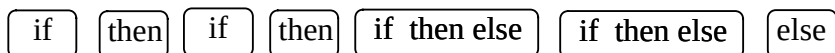
The pairing of **else** with a preceding **then** in the above sentence in PLs is done as follows :

if then if then if then else if then else else

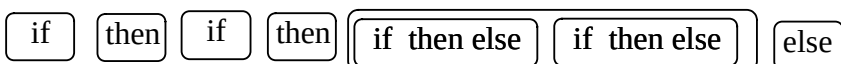
The pairing of the first else, in a left to right scan of the sentence, to its matching then, is shown as a larger box below.



The pairing of the second else to its matching then, is shown as the 2nd larger box.



The pairing of the last else to its match, is shown as the single large box, which encloses the earlier two boxes.



The pairing shown above is the unique interpretation for the given sentence. PL designers agree to the above interpretation. The matching rule can be specified as, “ in a left to right scan of the sentence, **else**

has to be paired with the closest preceding unmatched **then**.

The challenge now is to write a grammar that ensures the matching rule stated above.

Examination of the matching rule reveals that between any two consecutive **then** and **else**, only a complete “**if then else**” can occur, if any.

This crucial observation can be incorporated into a grammar that removes the ambiguity inherent in the earlier grammar.

S	→ matched-stmt unmatched-stmt
matched-stmt	→ if c then matched-stmt else matched-stmt other
unmatched-stmt	→ if c then S if c then matched-stmt else unmatched-stmt

Argue that the grammar for conditionals given above is indeed unambiguous.

Left recursion removal

We have seen that PL grammars usually have some recursive rules. We shall see later that left recursive grammars are unsuited for a class of parsing methods (top down parsing).

- A context free grammar is left recursive if there exists a nonterminal A, such that $A \Rightarrow^+ A\alpha$, for some α in $(N \cup T)^*$.

Simple case : Consider the rule $A \rightarrow A\alpha \mid \beta$. The grammar generates strings $\{\beta, \beta\alpha, \beta\alpha\alpha, \dots\}$. An equivalent grammar without left recursion is

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

It can be easily seen that the grammar above generates the same set of strings.

- If the left recursion is direct (or immediate), then even in a more general case such as

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

The rules above have m left recursive rules for A and also there are n alternatives of A that do not involve recursion. A left recursion free equivalent grammar is

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_m A' | \epsilon$$

The proof of equivalence is left to the reader.

The process of left recursion elimination does not work for all left recursive grammars. Write a few examples of left recursive grammars for which the suggested method does not work. Find the constraints under which the method will always work correctly.

Introduction To Parsing : Parsing Strategies

Let us begin with a formal definition of a parser.

A parser for a context free grammar G is a program P that when invoked with a string w as input, indicates that either

1. w is a sentence of G , i. e., $w \in L(G)$ (and may also give a parse tree for w), or
2. gives an error message stating that w is not a sentence (and may provide some information at or around the point of error).

Parsing Strategies

The two parsing strategies that we study are based on the following principles.

1. A parser scans an input token stream , from left to right, and groups tokens with the purpose of identifying a derivation, if one such exists.
2. In order to trace out such an unknown derivation, a parser makes use of the production rules of the grammar.
3. Different parsing approaches differ in the choice of derivation and/or the manner they construct a derivation for the input.
4. We have seen two standard derivations, viz., leftmost and rightmost, and also the way these can be represented through a parse tree. The two basic approaches to parsing can be easily explained using the concept of parse tree.
5. What are the possible ways in which a parse tree (or a tree data structure, in general) can be constructed?
 - Create the parse tree from the root downwards and expand the nodes till all leaves are reached, i. e., in a top down manner. Parsers of this type are known as **top-down** parsers.
 - Create the parse tree from leaves upwards to the root, i. e., in a bottom up fashion. Parsers which use this strategy belong to the family of **bottom-up** parsers.

6. There being a direct relation between parse trees and derivations, parsers which use anyone of the two parsing strategies can also be rephrased in terms of derivations.

We now describe in brief each parsing strategy in terms of its

- Basic principles : the derivation on which it is based, how the derivation is constructed, relevant issues and problems for designing a deterministic parser of this family.
- Manual construction of a parser : one or more parsing algorithms along with the data structures used, their limitations and strengths.
- Parser generators : how to automatically generate a parser from the cfg.

The details of the parsing algorithms shall be covered in subsequent lectures.

Principles of Top-Down Parsing

A top down parser creates a parse tree starting with the root, i. e., it starts a derivation from the start symbol of the grammar.

1. It could potentially replace any of the nonterminals in the current sentential form. However while tracing out a derivation , the goal is to produce the input string.
2. Since the input tokens are scanned from left to right , it will be desirable to construct a derivation that also produces terminals in the left to right fashion in the sentential forms.
3. A leftmost derivation matches the requirement exactly. A property of such a derivation is that there are only terminal symbols preceding the leftmost nonterminal in any leftmost sentential form.
4. The basic step in a top down parser is to find a candidate production rule (the one that could potentially produce a match for the input symbol under examination) in order to move from a left most sentential form to its succeeding left sentential form.
5. A top down parser uses a production rule in the obvious way - replace the lhs of the rule by one of its rhs.


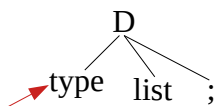
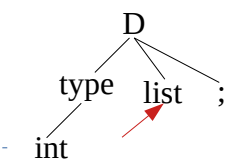
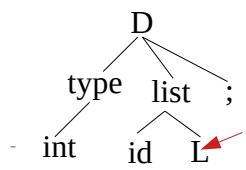
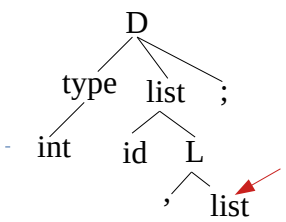
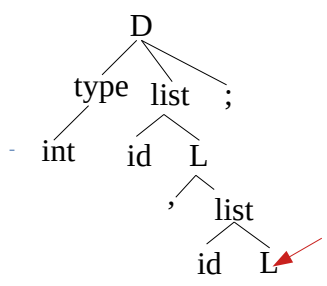
Example to Illustrate the Principles of Top-Down Parsing

Top-down parsing of the sentence “**int a, b;**” for the C declaration grammar, named as G1, given below.

R1 : D \rightarrow type list ;	R2 : type \rightarrow int
R3 : type \rightarrow float	R4 : list \rightarrow id L
R5 : L \rightarrow , list	R6 : L \rightarrow ϵ

The start symbol of the CFG is the nonterminal D. The rules have been numbered for ease of reference.

The processing of a top down parser over the given CFG and the input can be understood by tracing the moves of the parser. The current token in the input is highlighted and an arrow points to the leftmost terminal in the sentential form.

Input	Rule used with Explanation	Incremental Generation of Parse Tree
int a , b ;	None	Initial Parse Tree 
int a , b ;	Leftmost nonterminal is D R1 : D → type list ;	
int a , b ;	Leftmost nonterminal is type R2 : type → int int matches with the current token get next token; mark leftmost nonterminal	
int a , b ;	Leftmost nonterminal is list R4 : list → id L a matches with current token id get next token; mark leftmost nonterminal	
int a , b ;	Leftmost nonterminal is L R5 : L → , list , matches with current token , get next token; mark leftmost nonterminal	
int a , b ;	Leftmost nonterminal is list R4 : list → id L b matches with current token id get next token; mark leftmost nonterminal	

Input	Rule used with Explanation	Incremental Generation of Parse Tree
int a , b ;	Leftmost nonterminal is L $R6 : L \rightarrow \epsilon$ There are no leftmost nonterminal left in the parse tree The nexttoken ; matches with the ';' of rule R1	

At this stage the input string is matched and exhausted. The parsing is successful and the yield of the parse tree is exactly the input string.

Q. How would Top Down parsing proceed if we had used the following equivalent grammar, say G2, instead ?

$R1 : D \rightarrow \text{type list ;}$
 $R2 : \text{type} \rightarrow \mathbf{int}$
 $R3 : \text{type} \rightarrow \mathbf{float}$
 $R4 : \text{list} \rightarrow \mathbf{id} , \text{list}$
 $R5 : \text{list} \rightarrow \mathbf{id}$

Processing CFG rules to generate some useful information about its nonterminals. Let us consider the earlier grammar.

$R1 : D \rightarrow \text{type list ;}$
 $R2 : \text{type} \rightarrow \mathbf{int}$
 $R3 : \text{type} \rightarrow \mathbf{float}$
 $R4 : \text{list} \rightarrow \mathbf{id} L$
 $R5 : L \rightarrow , \text{list}$
 $R6 : L \rightarrow \epsilon$

- A nonterminal including the start nonterminal can derive several sentential forms using the grammar rules. The nonterminals are {D, type, list, L}
- Consider the following derivations from D

$D \Rightarrow \text{type list ;} \Rightarrow \mathbf{int} \text{ list ;}$ $D \Rightarrow \text{type list ;} \Rightarrow \mathbf{float} \text{ list ;}$

While D can derive many more sentential forms, it can be seen that the first terminals that can appear in any sentential form (leftmost, rightmost or an arbitrary one) derivable from D are just two, namely {int, float}.

Similar information about the non-terminals of G1 can also be obtained. This information about a nonterminal, say A, is denoted by FIRST(A), and is very useful in parsing. For example, when the first token of a input, say “short”, is checked for syntactic validity with respect to G1, a top down parser will try to match “short” with FIRST(D) = {int, float} and since it fails will declare an error.

It appear at a first glance that to construct FIRST(A), for all A in the set of nonterminals, one would have to generate all the sentential forms. Fortunately that is not the case and these sets can be easily constructed using an algorithm that processes the rules iteratively.

Construction of FIRST() sets

The set of terminals that can ever appear in any sentential form derivable from a string α , $\alpha \in (N \cup T)^*$, is denoted by FIRST(α).

Formally, this is defined as $FIRST(\alpha) = \{ a \in T^* \mid \alpha \Rightarrow^* a \beta \text{ for some } \beta \}$.

It follows that $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in FIRST(\alpha)$

Some obvious facts about FIRST information are the following :

1. For a terminal a, $FIRST(a) = \{ a \}$
2. For a nonterminal A, if $A \rightarrow \epsilon$, then $\epsilon \in FIRST(A)$
3. For a rule $A \rightarrow X_1 X_2 \dots X_r$

$$FIRST(A) = FIRST(A) \cup_k (FIRST(X_k)); 1 \leq k \leq r; \epsilon \in X_i; 1 \leq i \leq k-1;$$

The expression above says that the contributions from the FIRST sets of X_2, X_3, \dots, X_{k-1} also need to be computed if all their preceding nonterminals can derive ϵ . For instance, the terminals in $FIRST(X_5)$ are to be added to $FIRST(A)$, only if, $\epsilon \in FIRST(X_1)$ and $\epsilon \in FIRST(X_2)$ and $\epsilon \in FIRST(X_3)$ and $\epsilon \in FIRST(X_4)$.

An Algorithm that computes FIRST sets of nonterminals for a general CFG follows.

Algorithm for FIRST sets

Input : Grammar $G = (N, T, P, S)$

Output : $FIRST(A)$, $A \in N$

Method :

begin algorithm

for $A \in N$ do $FIRST(A) = \Phi$;

 change := true;

 while change do

 { change := false;

 for $p \in P$ such that p is $A \rightarrow \alpha$ do

 { newFIRST(A) := $FIRST(A) \cup Y$; where Y is $FIRST(\alpha)$ from definition;

 if newFIRST(A) \neq FIRST(A) then

 { change := true ; FIRST(A) := newFIRST(A);}

 }

 }

end algorithm

Illustration of FIRST() Set Computation :

R1 : $D \rightarrow \text{type list}$; R2 : $\text{type} \rightarrow \text{int}$

R3 : $\text{type} \rightarrow \text{float}$ R4 : $\text{list} \rightarrow \text{id}$ L

R5 : $L \rightarrow , \text{list}$ R6 : $L \rightarrow \epsilon$

Nonterminal	Initialization	Iteration 1	Iteration 2	Iteration 3
D	Φ	Φ	{int float}	{int float}
type	Φ	{int float}	{int float}	{int float}
list	Φ	{id}	{id}	{id}
L	Φ	{ ϵ ,}	{ ϵ ,}	{ ϵ ,}
change	-	True	True	False

The working of the algorithm can be explained as follows. The initialization step is obvious. In the first iteration, rule R1 is used for $FIRST(D) = FIRST(\text{type list} ;) = FIRST(\text{type}) = \Phi$

$FIRST(\text{type}) = FIRST(\text{int}) = \{\text{int}\}$ using R2 and using R3 $FIRST(\text{type}) = FIRST(\text{float}) = \{\text{int float}\}$.

Since $FIRST(\text{type})$ has changed from Φ to {int float}, change is set to True. Similarly in the second

iteration, $\text{FIRST}(D)$ changes due to which change is again set to True. All the $\text{FIRST}()$ sets in iteration 3 remain the same as they were at the end of iteration 2, which causes change to be assigned False and the algorithm terminates with the $\text{FIRST}()$ sets as given at the end of iteration 3.

Q. Construct a derivation that generates a sentential form justifying the presence of a terminal in the $\text{FIRST}(A)$ for any A .

Q. What is the worst complexity of the algorithm in terms of the quantities present in the grammar?

Another Useful Information From the CFG

Another information that is found to be very useful in several parsing methods (but not all) is also based on sentential forms that are generated by the grammar rules.

- Consider the following derivations from D using grammar G_1

$R1 : D \rightarrow \text{type list ;}$	$R2 : \text{type} \rightarrow \text{int}$
$R3 : \text{type} \rightarrow \text{float}$	$R4 : \text{list} \rightarrow \text{id L}$
$R5 : L \rightarrow , \text{list}$	$R6 : L \rightarrow \epsilon$

$D \Rightarrow \text{type list ;} \Rightarrow \text{int list ;} \quad D \Rightarrow \text{type list ;} \Rightarrow \text{type id L ;}$

$D \Rightarrow \text{type list ;} \Rightarrow \text{float list ;}$

$\text{list} \Rightarrow \text{id L} \Rightarrow \text{id , list} \Rightarrow \text{id , id L} \Rightarrow \dots$

- The information that is sought is about the terminals that appear immediately to the right of a nonterminal in any sentential form of G_1 . Such information is named as $\text{FOLLOW}(A)$ for a nonterminal A and gives the terminal symbols that immediately follow A in some sentential form of G_1 .
- For instance, using $R1$ it can be seen that $\text{FOLLOW}(\text{list})$ contains ‘;’ similarly from the derivation sequence, “ $D \Rightarrow \text{type list ;} \Rightarrow \text{type id L ;}$ ” we find that $\text{FOLLOW}(L)$ contains ‘;’. It is intended to construct $\text{FOLLOW}(A)$ for all A in N .

Rules for construction of $\text{FOLLOW}()$ sets are the following.

Rule 1 : For the start symbol of the grammar, $\$ \in \text{FOLLOW}(S)$. The rationale for this special rule is that it is essential to know when the entire input has been examined. A special marker, $\$$, is appended to be actual input. Since S is the start symbol, only the end of input marker can follow S .

Rule 2 : If $A \rightarrow \alpha B \beta$ is a grammar rule, then $\text{FOLLOW}(B) \cup = \{\text{FIRST}(\beta) - \epsilon\}$

Rule 3 : If either $A \rightarrow \alpha B$ is a rule or if $A \rightarrow \alpha B \beta$ is a rule and $\beta \Rightarrow^* \epsilon$ (that is, $\epsilon \in \text{FIRST}(\beta)$), then $\text{FOLLOW}(B) \cup = \text{FOLLOW}(A)$

The rule says that under the given conditions, whatever follows A also follows B.

- These rules can be applied to write an algorithm that constructs $\text{FOLLOW}(A)$ for all A in N iteratively similar to the algorithm for construction of FIRST sets.

G1 is reproduced below for ready reference.

$R1 : D \rightarrow \text{type list ;}$ $R2 : \text{type} \rightarrow \text{int}$
 $R3 : \text{type} \rightarrow \text{float}$ $R4 : \text{list} \rightarrow \text{id L}$
 $R5 : L \rightarrow \text{, list}$ $R6 : L \rightarrow \epsilon$

- Rule 1 applied to R1 gives $\text{FOLLOW}(D) = \{\$ \}$.
Rule 2 applied to R1 gives $\text{FOLLOW}(\text{type}) \cup = \text{FIRST}(\text{list}) = \{\text{id}\}$;
 $\text{FOLLOW}(\text{list}) \cup = \text{FIRST}\{;\} = \{;\}$
- No Rule applies to R2 and R3.
- Rule 3 applied to R4 gives $\text{FOLLOW}(L) \cup = \text{FOLLOW}(\text{list}) = \{;\}$
- Rule 3 applied to R5 gives $\text{FOLLOW}(\text{list}) \cup = \text{FOLLOW}(L) = \{;\}$
- Processing of the $\text{FOLLOW}()$ sets iteratively shows that the information converges at the end of iteration 2.

A in N	FIRST()	Initialization	Iteration 1	Iteration 2
D	{int float}	{ \$ }	{ \$ }	{ \$ }
type	{int float}	Φ	{id}	{id}
list	{id}	Φ	{;}	{;}
L	{ ϵ ,}	Φ	{;}	{;}
change		NA	True	False

Q. Construct a derivation that generates a sentential form justifying the presence of a terminal in the $\text{FOLLOW}(A)$ for any A.

Q. Write an algorithm for construction of $\text{FOLLOW}()$ for all nonterminal symbols of a CFG.

Q. What is the worst complexity of your algorithm in terms of the quantities present in the grammar?

****** End of Document ******

Lexical Analysis

1. Motivation

Consider an input C program. This is the way an editor will show it.

```
main ()
{
    int i,sum;
    sum = 0;
    for (i=1; i<=10; i++);
    sum = sum + i;
    printf("%d\n",sum);
}
```

The same program as the compiler sees it initially, as a continuous stream of characters.

```
main_()↵{↵_int_i,sum;↵_sum=_0;↵_
for_(i=1;_i<=10;i++);_sum=_sum+_i;↵_
printf("%d\n",sum);↵}
```

Where the symbol _ indicates a blank space and ↵ indicates a newline character.

2. Partitioning the Input

How do we make the compiler see the way that we interpret it, not as a sequence of characters but as a collection of meaningful entities?

Discover the structure in the input.

Step 1:

a. Break up this string into ‘words’–the smallest logical units.

```
main ( ) { int i , sum ; sum = 0 ; for ( i = 1 ; i <= 10 ; i ++ ) ; sum = sum + i ; printf ( "%d\\n" , sum ) ; }
```

Each shaded rectangular box denotes a lexeme of the program. The result is a sequence of lexemes (or tokens).

b. Clean up – remove the blank space `_` and the newline character `↵`.

The resulting sequence after this operation is shown below.

```
main ( ) { int i , sum ; sum = 0 ; for (
i = 1 ; i <= 10 ; i ++ ) ; sum = sum + i
; printf ( "%d\n" , sum ) ; }
```

This is **lexical analysis** or **scanning**.

Important Terms : Lexemes, Tokens and Patterns

Definition: Lexical analysis is the operation of dividing the input program into a sequence of lexemes (tokens).

Let us learn to distinguish between

- **lexemes** – smallest logical units (words) of a program.

Examples – i, sum, for, 10, ++, "%d\n", <=.

- **tokens** – sets of similar lexemes.

Examples –

identifier = {i, sum, buffer, . . . }

int constant = {1, 10, . . . }

addop = {+, -}

Things that are not counted as lexemes

- white spaces – tab, blanks and newlines
- comments

However even these too have to be detected and stripped off the input and ignored.

Q. How does one describe the lexemes that make up the token identifier?

Even for an identifier, there are variants in different languages. We need to know the description of an identifier in the Programming Language (PL) of interest. C is the language for our example program.

Such descriptions are called **patterns**. The description may be informal or formal. An informal description follows.

- a string of alphanumeric characters. The first character is an alphabet.
- a string of alphanumeric characters in which the the first character is an alphabet. There is an additional constraint that the length of an identifier can be at most 31.
- a string of alphanumeric or underline (`_`) characters in which the the first character is an alphabet or an underline. The length of an identifier is restricted cannot exceed 31. However, an identifier with `> 31` characters, the first 31st character are retained and the remaining are ignored.

Basic concepts and issues

A pattern in the context of lexical analysis is used to

- specify a token precisely, and
- build a recognizer from such specifications

Example – tokens in Java

1. Identifier: A Javaletter followed by zero or more Javaletterordigits.

A Javaletter includes the characters a-z, A-Z, `_` and `\$`.

2. Constants: The following are all valid constants.

2.1 Integer Constants

- Octal, Hex and Decimal
- 4 byte and 8 byte representation

2.2 Floating point constants

- float - ends with f
- double

2.3 Boolean constants – true and false

2.4 Character constants – 'a', '\u0034', '\t'

2.5 String constants – "", "\", "A string"

2.6 Null constant – null

3. Delimiters: () { } [] ; . and ,

4. Operators: =, >, < . . . >>, >=

5. Keywords: abstract, boolean . . . volatile, while

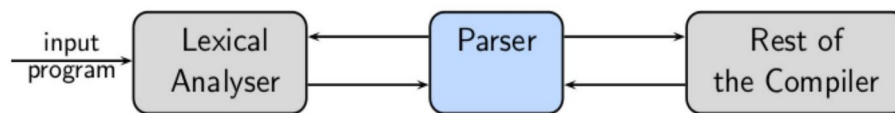
Exercise : Read up the specification of an identifier in language C and write this informally on the same lines as given above.

Context Of A Lexical Analyser

Where does a lexical analyser fit into the rest of the compiler?

- The lexical analyser is used primarily by the parser (part of a compiler that performs syntax analysis). When the parser needs the next token, it calls the lexical analyser.
- Instead of analysing the entire input string, the lexical analyser sees enough of the input string to return a single token to the parser.
- Thus lexical analysis and parsing, which are distinct phases because they perform distinct are not different passes of a compiler, typically belong to the same pass.

EXAMPLE



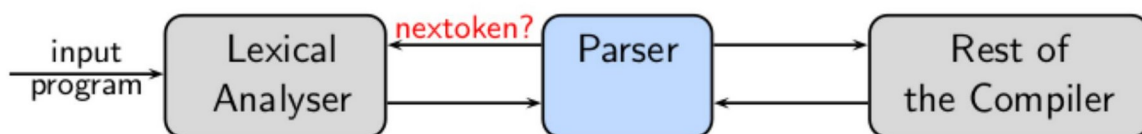
The Lexical Analyzer (or scanner) and the Parser (or Syntax Analyzer) work in tandem as shown below. Consider the initial part of the input program :

```
main ()
```

```
{
```

```
    int i,sum;
```

- Parser asks the lexical analyzer to supply a token.

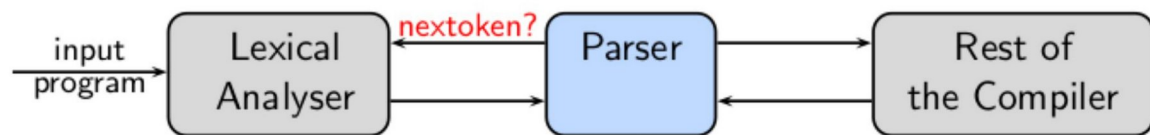
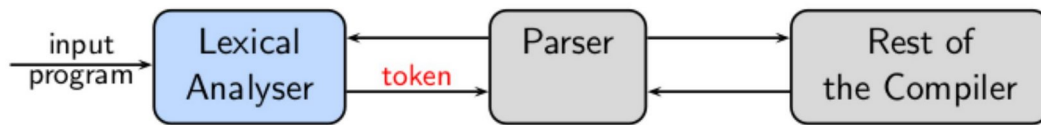


- The scanner identifies a token and returns the same to the parser. In this case, scanner returns : <identifier, "main">. The remaining input is now :

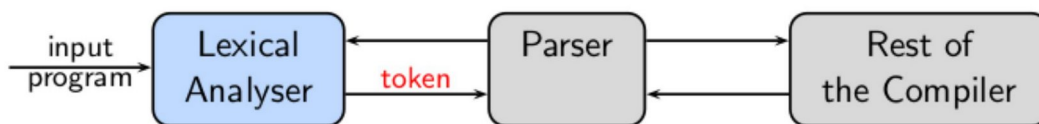
```
)
```

```
{
```

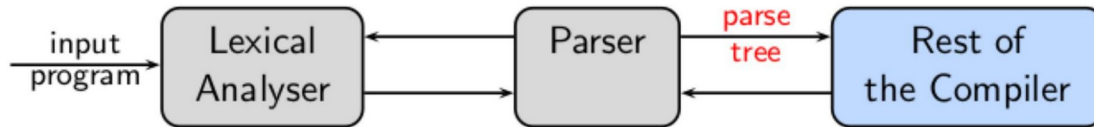
```
    int i,sum;
```



- After consuming “main”, the parser again asks the scanner for the next token, as seen earlier.
- The scanner returns the token “(“ this time with the remaining input as :
)
 {
 int i,sum;



Such communication between the parser and continues till the entire program has been processed. At this point the parser generates a parse tree as its output and communicates the tree with the rest of the compiler.



Note that the above description is a simple model to illustrate the nature of interaction between the parser and the scanner. In a real compiler, the interaction between the parser and the rest of the compiler is more involved as we shall see later in this course.

Token Attributes

Apart from the token itself, the lexical analyser also passes other information regarding the token. These items of information are called token attributes.

lexeme	Token type	token attribute
3	const	3
x1	identifier	x1
if	keyword	(index in keyword list)
>=	relop	>= (or index in list of relops)
;	delimiter	;

Creating a Lexical Analyzer

Two approaches:

1. **Hand code the lexical analyser** – that is write it manually This is of historical interest now. A human may be able to write more efficient code but correctness always remains a concern.

2. : generate the lexical analyser from a formal des **Use a lexical analyser generator** cription of the tokens of the language and their attributes.

- The generation process is faster to yield a working scanner.
- Less prone to errors.

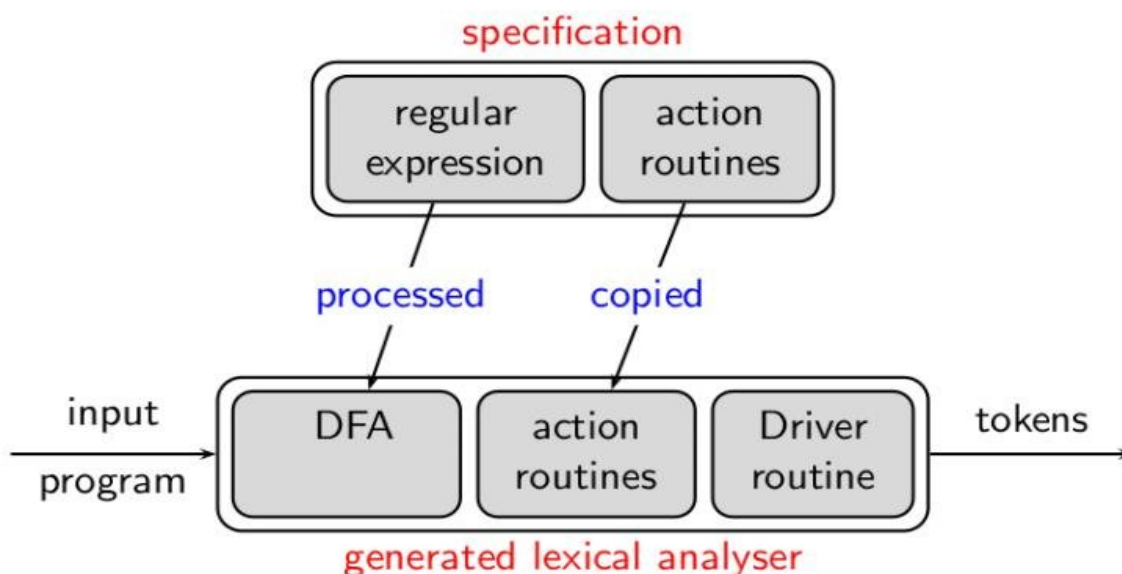
Automatic Generation of Lexical Analysers

Inputs to the lexical analyser generator:

- A specification of the tokens of the source language, which involves :
 - a regular expression describing each token, and
 - a code fragment describing the action to be performed, on identifying each token.

The generated lexical analyser consists of:

- A deterministic finite automaton (DFA) constructed from the token specification.
- A code fragment (a driver routine) which can traverse any DFA.
- Code for the action specifications when a token is recognized.



- The first row indicates the effort at the user level. This involves writing a token in terms of regular expression and state the list of actions to be executed when that token is detected in the input.
- The scanner generation algorithm processes the regular expressions and constructs a DFA first and then minimizes the DFA.
- It copies the user defined actions and converts them into functions that can be called when required.
- A generic driver routine is included, whose purpose is to read a token character by character and use the DFA for recognition. When a final state is encountered, the function corresponding specified action is executed by the driver.
- The components in the bottom row constitutes the lexical analyzer that is automatically generated. It is a one time effort.

The generated lexical analyzer (or scanner) scans a stream of input, recognizes a token and communicates the token to the parser as can be seen in the last row of the above figure.

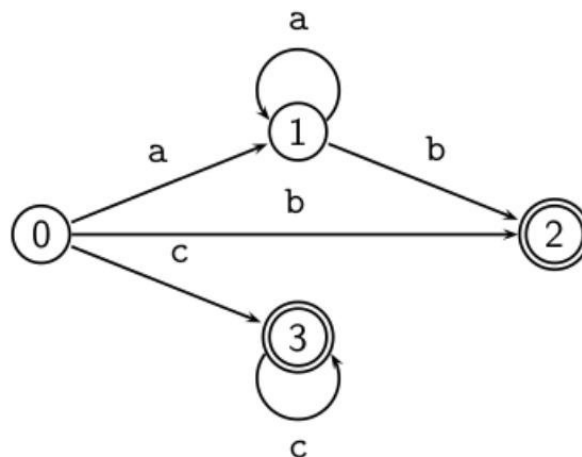
The entire process is illustrated through a working example.

EXAMPLE : Consider a simple language with two tokens along with the associated actions.

Pattern	Action
a^*b	{ printf("Token 1 Recognized \n"); }
c^+	{ printf("Token 2 Recognized \n"); }

The Alphabet is $\Sigma = \{a, b, c\}$ and the $\{*, +\}$ are the usual regex operators.

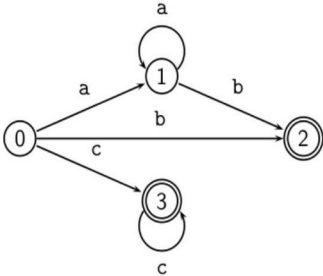
The scanner generation software constructs a dfa from the regex, such as the one shown below. There exists an algorithm for directly constructing a dfa from regular expressions (instead of going round through a NFA). This algorithm is generally not a part of FLAT curriculum and hence we shall discuss it in this course.



The three components that are a part of the generation tool are :

- ▶ a DFA
- ▶ a driver routine
- ▶ action functions

Let us consider these parts together

DFA	Action functions
 <pre> graph LR 0((0)) -- a --> 1((1)) 0 -- b --> 2(((2))) 0 -- c --> 3(((3))) 1 -- a --> 1 1 -- b --> 2 2 -- a --> 2 2 -- b --> 2 3 -- c --> 3 </pre>	<p>The actions are converted into a function</p> <pre> function action(state) { case state 2 : { printf("Token 1 Recognized \n"); 3 : { printf("Token 2 Recognized \n"); } } </pre>
Driver routine	Working : input & output
<pre> function nexttoken() { state = 0; c = nextchar(); while (valid(nextstate[state, c])) { state = nextstate[state, c]; c = nextchar(); } if (!final(state)) {error message; return;} else { unput(c); action(state); return;} } </pre>	<p>Input : aabbaccabc</p> <p>The first character in the input is highlighted for clarity.</p>

Data structure and support functions used by the driver routine :

- `nextstate[state s, char c]` is a data structure (a realization of the DFA), which given a state `s` and the character '`c`' in the input, gives the next state.
- `nextchar()` : returns the next character from the input stream
- `final(state s)` : returns true if the argument state `s`, is a final state else returns false
- `unput(c)` : returns the character '`c`' back to input stream. Determine the reason for including this function.
- `action(s)` : executes the action provided by the user on the state `s`; note that only a final state has an associated action corresponding to a token that it has recognized in state `s`.

The driver routine starts with start state 0.

The input string is : `aabbaccabc`

The configuration of the system at the end of each iteration of the while loop is tabulated below.

Observe that each iteration of the while loop starts with a given state `s` and a character `c`.

On entry to the loop body, a change of state is executed and the next character in the input is read.

The 1st column gives the iteration number. The 2nd and 3rd columns denote the current state and the next character in the input before entering the loop body. The 4th and 5th columns give the changes to state and the `nextchar` caused by the statements in the body of the loop. The output of the generated scanner is highlighted in green as also the final states.

Examine the table given below that shows how the generated scanner processes the input : `aabbaccabc`

For a better understanding, show the effect of each row of the table by drawing the dfa and the state reached, the input symbols consumed and the input that remains to be scanned yet.

Iter	state s	nextchar c	nextstate [s,c]	nextchar c	Rest of input	Move / Output
1	0	a	1	a	bbaccabc	Move to state 1
2	1	a	1	b	baccabc	Remain in state 1
3	1	b	2	b	accabc	Move to state 2
4	2	b	–	b	baccabc	unput(b); return Token 1 Recognized
5	0	b	2	a	ccabc	Restart; move to state 2
6	2	a	–	a	accabc	unput(a); return Token 1 Recognized
7	0	a	1	c	cabc	Restart; move to state 1
8	1	c	–	c	ccabc	unput(c); Error; skip a ; return
9	0	c	3	c	abc	Restart; move to state 3
10	3	c	3	b	c	Remain in state 3
11	3	b	–	b	bc	unput(b); return Token 2 Recognized
12	0	b	2	c	NIL	Restart; move to state 2
13	2	c	–	c	c	unput(c); return Token 1 Recognized
14	0	c	3	–	NIL	Restart; move to state 3
15	3	NIL	–	–	NIL	Return; Token 2 Recognized scanner terminates

In conclusion we have shown that the DFA, the driver routine and the action routines, all taken together constitute the lexical analyser. Further, given specifications of patterns and actions by the user

- patterns are used to automatically construct a final DFA
- actions that are supplied as part of specification are converted into callable functions
- a generic driver routine is written and included as a one time effort that is common to all generated lexical analyzers

LEXICAL ERRORS

Lexical errors are essentially of two kinds.

1. Lexemes whose length exceed the bound specified by the language.

For example, In Fortran, an identifier more than 7 characters long is a lexical error.

Most languages have a bound on the precision of numeric constants.

A constant whose length exceeds this bound is a lexical error.

2. Illegal characters in the program

The characters ~, & and @ occurring in a Pascal program (but not within a string or a comment) are lexical errors.

3. Unterminated strings or comments

In C / C++, a string that does not have the delimiter “ at the end of the string is an unterminated string

multi-line comment that does not end with “*/”

Handling of Lexical Errors

The action taken on detection of a lexical error are:

Issue an appropriate error message indicating the error to the program writer.

- Error of the first type – the entire lexeme is read and then truncated to the specified length
- Error of the second type – skip illegal character.

Alternative is to pass the illegal character to the parser which has better knowledge of the context in which error has occurred. There exist more possibilities of recovery replacement instead of deletion.

Error of the third type – process till end of file and then issue error message if the terminating string is not found.

An Efficient Algorithm for Direct Construction of DFA from Regular Expressions

Recapitulate the Basics of Regular Expressions from FLAT.

- An Alphabet Σ is any finite set of symbols.
- A string over an alphabet is a finite (possibly zero) sequence of symbols from Σ .
- A Language L is any set of strings over an Alphabet Σ , formally expressed as $L \subseteq \Sigma^*$; where $L = \bigcup_{i=0}^{\infty} L^i$
- The empty set Φ is the smallest language over any Σ while the largest is Σ^*
- Operations on languages : union, intersection, concatenation
- The basic regular expression operators and the language denoted by them is given below.

Expression	Describes	Language	Example
ϵ	Empty string	$\{\epsilon\}$	
c	Any non-metalinguistic character c	$\{c\}$	a
r^*	Zero or more r 's	$L(r)^*$	a^*
r^+	One or more r 's	$L(r)^+$	a^+
$r1 \cdot r2$	$r1$ followed by $r2$	$L(r1) \cdot L(r2)$	$a \cdot b$ or ab
$r1 \mid r2$	$r1$ or $r2$	$L(r1) \cup L(r2)$	$a \mid b$
(r)	r	$L(r)$	$(a \mid b)$

Table 1 : Regular Expression Operators and Underlying Languages

Like the operators in programming languages, the regular expression operators also have precedence and associativity operators as given below.

	Regular expression operators with precedence values			
Operator	Parentheses ()	Closure * Reflexive closure +	Concatenation •	Alternate
Precedence	4	3	2	1
Associativity	Left	Right	Left	Left

Table 2 : Properties of Regular Expression Operators

Consider the following regular expression notation for the two tokens used earlier.

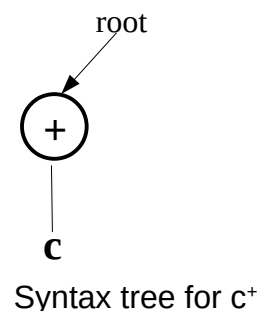
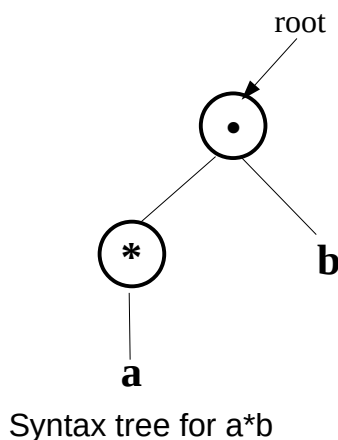
a^*b Token 1
 c^+ Token 2

First we combine all the regular expressions into a single regex as shown below.

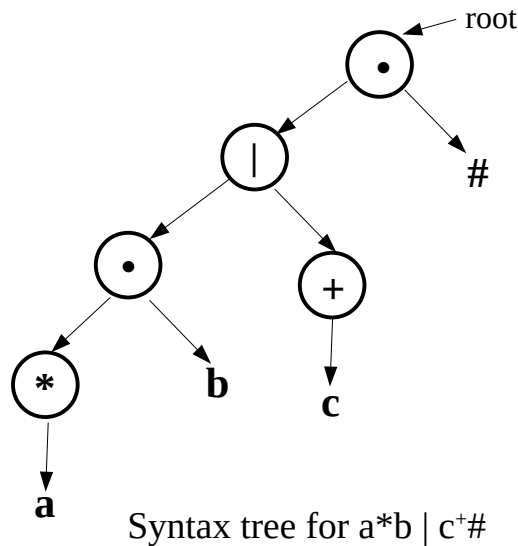
$a^*b \mid c^+$

We construct a syntax tree for the above regex that honours all the operator properties.

The syntax tree for a^*b is same as $a^* \cdot b$; since $*$ has higher precedence than \cdot , the tree is as drawn below. Similarly the syntax tree for c^+ is also drawn.



Combining the two trees with '|' and appending an endmarker symbol (#) yields the tree for the complete regex, rooted at •



It should be clear from the construction process outlined above that given a set of token specifications, one can combine all the regexes to construct a single syntax tree.

We know that a regex denotes a set of strings from the underlying alphabet.

For example, the regex **a** denotes the singleton string {a}.

The regex **a*** denotes the infinite set of strings { ϵ , **a**, aa, aaa, aaaa, }.

The regex **a*b** denotes the infinite set of strings {**b**, ab, aab, aaab, aaaab, ... }.

The regex **c+** denotes the infinite set of strings {**c**, cc, ccc, }.

Examining the regex, $a^*b \mid c^+$, and its associated strings reveal that the first character in any string belonging to the language, $L(a^*b \mid c^+)$, is one of 'a', 'b' or 'c' (highlighted). Since the same character may appear more than once in a regular expression, such as, $(ab)^+ \mid (a \mid b)^*b$, indicating a character symbol by name is ambiguous.

The set of strings of $(ab)^+$ is {ab, abab,}.

The set of strings of $(a \mid b)^*b$ is {b, ab, bb, aab, abb, bab,}.

For the regex, $(ab)^+ \mid (a \mid b)^*b$, the first character in $L((ab)^+ \mid (a \mid b)^*b)$ can be written as either a or b, which is misleading and wrong.

Instead of the character symbol, if we can replace its position index, a precise information is obtained. All the alphabet symbols have been given a unique index value.

$$(a\ b)^+ \mid (a\ \mid\ b)^* b$$

1 2 3 4 5

The set of strings of $(ab)^+$ is {ab, abab,}.

12 1212

The set of strings of $(a \mid b)^*b$ is {b, ab, bb, aab, abb, bab,}.

5 35 45 335 345 435

For the regex, $(ab)^+ \mid (a \mid b)^*b$, the first character in $L((ab)^+ \mid (a \mid b)^*b)$ can now be written as {1, 3, 4, 5} because of the highlighted strings above.

Using the position of alphabet symbols, an algorithm was designed that uses the syntax tree as input and directly constructs the dfa using the position of symbols in the input and the semantics of the regex operators.

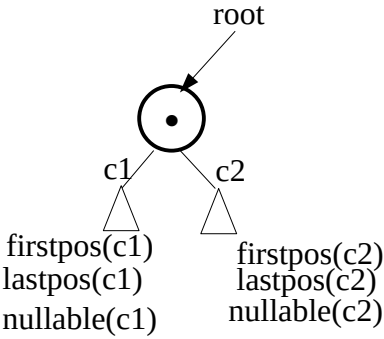
A few terms and definitions are needed. It is assumed that a syntax tree for the regex has been created. The internal nodes of the tree are the regex operators and the the alphabet symbols are place at the leaf nodes.

Four functions are used in the construction process. Consider a syntax tree rooted at a node, say x.

- $nullable(x)$ indicates whether the tree rooted at x can generate the null string ϵ in which case the function returns True otherwise it returns False.
- $firstpos(x)$ denotes the set of characters, by position, that are the first characters in any string generated from x
- $lastpos(x)$ denotes the set of characters, by position, that are the last characters in any string generated from x
- $followpos(i)$ denotes the set of characters, by position, that can follow the character at position i in any string generated from th regex.

Let Σ be the underlying alphabet. The rules for construction of the sets, $\text{firstpos}()$, $\text{lastpos}()$ and $\text{followpos}()$ are summarized in the following table. It is assumed that $\text{followpos}(i)$ is initialized to \emptyset , \forall positions labeled i . The table entry for $c_1 \bullet c_2$ is explained below.

Concatenation Operator \bullet : Consider a tree rooted at \bullet with two subtrees rooted at c_1 and c_2 respectively. It is assumed that $\text{nullable}()$, $\text{firstpos}()$ and $\text{lastpos}()$ are already known for both c_1 and c_2 .

	<p>The symbols that happen to be at the first positions of $\text{firstpos}(\bullet)$ are definitely those positions that are present in $\text{firstpos}(c_1)$.</p> <p>However in case ϵ belongs to $\text{firstpos}(c_1)$, then all the positions in $\text{firstpos}(c_2)$ are also included in $\text{firstpos}(\bullet)$. The calculations for $\text{nullable}(\bullet)$, $\text{firstpos}(\bullet)$ and $\text{lastpos}(\bullet)$, given in the table, in terms of their children are based on these relations.</p>
--	--

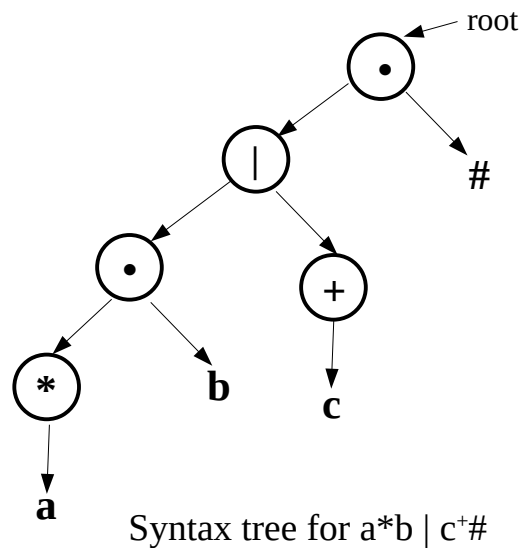
Rules for computation of the four functions

<i>node n</i>	<i>nullable(n)</i>	<i>firstpos(n)</i>	<i>lastpos(n)</i>	<i>followpos(i)</i>
Leaf labeled ϵ	true	\emptyset	\emptyset	
Leaf labeled with $a \in \Sigma$ at position i	false	$\{i\}$	$\{i\}$	
$c_1 \mid c_2$	$\text{nullable}(c_1)$ or $\text{nullable}(c_2)$	$\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$	$\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$	
$c_1 \bullet c_2$	$\text{nullable}(c_1)$ and $\text{nullable}(c_2)$	if $\text{nullable}(c_1)$ then $\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$	if $\text{nullable}(c_2)$ then $\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$ else $\text{lastpos}(c_2)$	if $i \in \text{lastpos}(c_1)$ then $\text{followpos}(i) += \text{firstpos}(c_2)$

<i>node n</i>	<i>nullable(n)</i>	<i>firstpos(n)</i>	<i>lastpos(n)</i>	<i>followpos(i)</i>
		else firstpos(c_1)		
c^*	true	firstpos(c)	lastpos(c)	if $i \in \text{lastpos}(c)$ then followpos(i) += firstpos(c)
c^+	nullable(c)	firstpos(c)	lastpos(c)	if $i \in \text{lastpos}(c)$ then followpos(i) += firstpos(c)

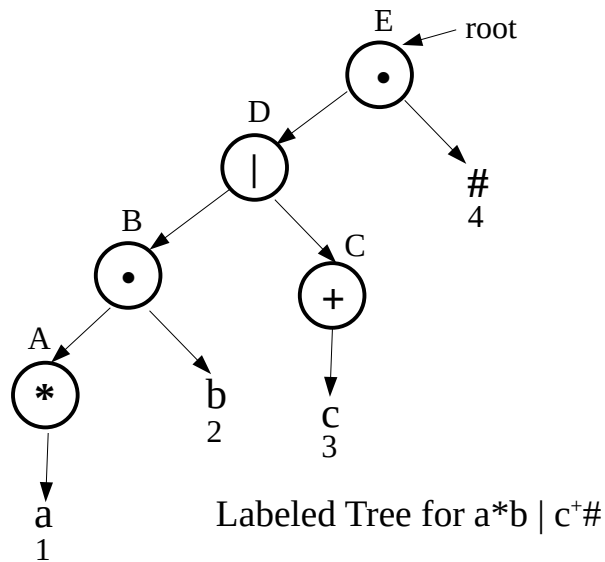
From the table above, it can be seen that only 3 operators, $\{*, +, \bullet\}$ contribute to the followpos() information, while the other operators '|' and the leaf nodes do not affect followpos().

The syntax tree for the regex is used to illustrate the computation of the 4 functions by making a bottom up traversal of the tree.



All the leaf nodes are numbered from 1 to 4 while the internal nodes are labeled by upper case alphabets.

The tree with all its nodes with labels are shown below.

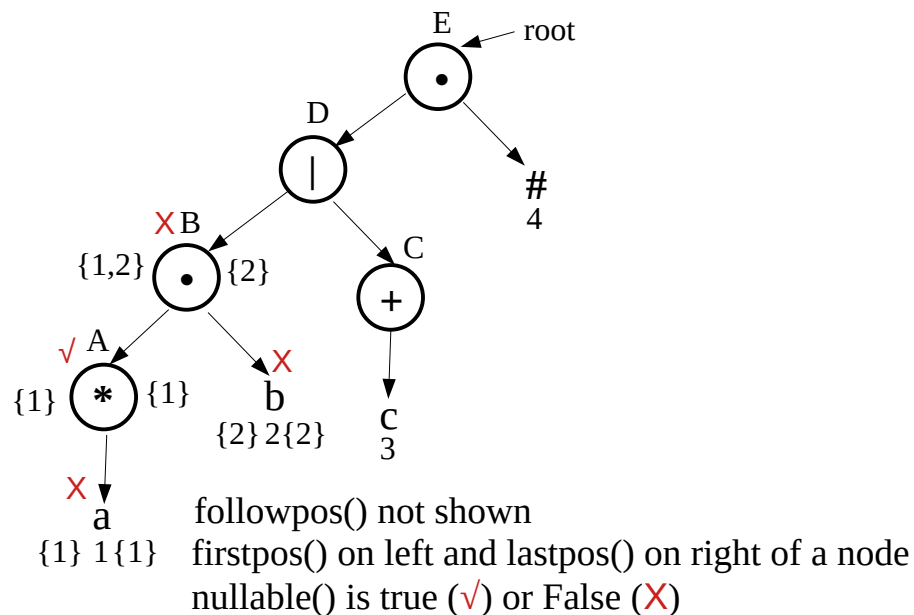


The symbol \times is used to denote that the node is NOT nullable while the symbol \checkmark is used to denote that the node is nullable. The firstpos() and lastpos() are placed at the left and right of a given node.

1. Start with the leaf node 1. This is not nullable, so nullable(1) = False, and firstpos(1) = lastpos(1) = {1}
2. For the internal node labeled A, nullable(A) is True, firstpos(A) = firstpos(1) = {1} and lastpos(A) = lastpos(1) = {1}. Since A has operator *, it also contributes to followpos(). Since 1 belongs to lastpos(1), we create followpos(1) = {1}, where the {1} in the RHS is because of firstpos(1).
3. The leaf node 2 is NOT nullable and firstpos(2) = lastpos(2) = {2}
4. The internal node B has \bullet operation whose left and right children have all the 3 sets computed. The nullable(B) is False because the right child is not nullable. firstpos(B) = firstpos(A) \cup firstpos(2) = {1, 2}
lastpos(B) = lastpos(2) = {2}.

Since B has \bullet , it contributes to `followpos()`. For all elements in `lastpos(c1)`, which turns out to be `lastpos(1) = {1}`, we have to add `firstpos(c2)`, that is 2, to the existing `followpos(1)`, which now becomes `{1, 2}`.

Node / leaf	Symbol	nullable (node)	firstpos()	lastpos()	followpos()
1	a	False (✗)	{1}	{1}	{1} ∪ {2}
A	*	True (✓)	{1}	{1}	
2	b	False (✗)	{2}	{2}	
B	•	False (✗)	{1,2}	{2}	



Labeled Tree for $a*b \mid c^{\#}$: values at subtree rooted at B

Note that $\text{followpos}(3) = \{3\}$ due to $+$ in node C. Further the \bullet operation at root E, results in assigning $\text{followpos}(\text{lastpos}(D))$ to $\text{firstpos}(4)$, that is updating the $\text{followpos}()$ at 2 and 3 as indicated by, $\text{followpos}(2) \cup = \{4\}$, and $\text{followpos}(3) \cup = \{4\}$. Continuing along the same lines, when the entire syntax has been processed, the final information of all the 4 sets are as shown in the following table.

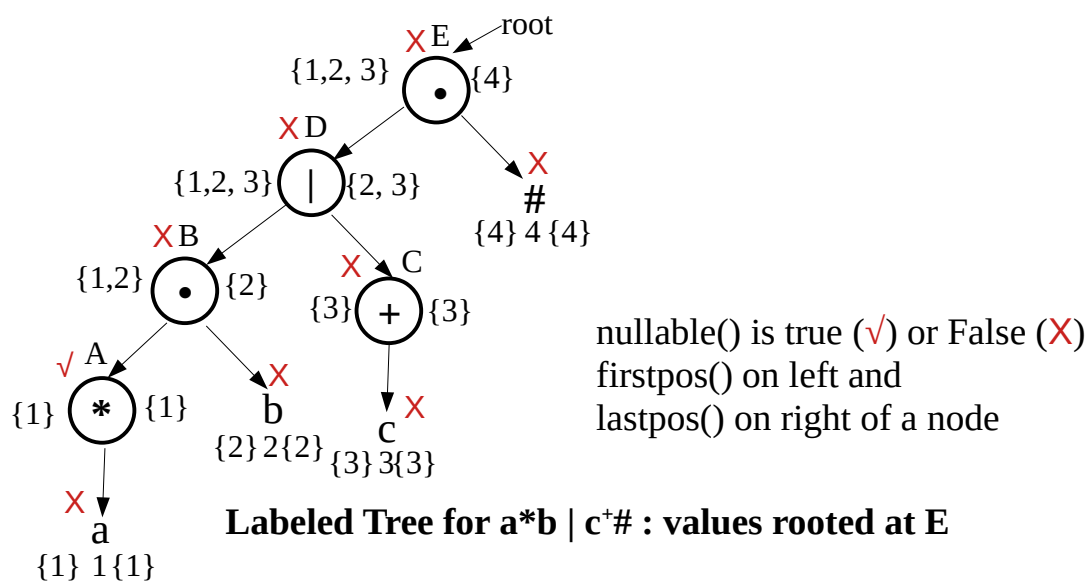
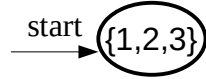


Table 3 : Values of 4 functions at each node of the Tree

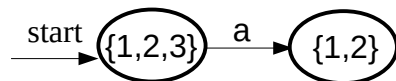
Node / leaf	Symbol	nullable (node)	firstpos()	lastpos()	followpos()
1	a	False	{1}	{1}	$\{1\} \cup \{2\}$
A	*	True	{1}	{1}	
2	b	False	{2}	{2}	{4}
B	•	False	{1,2}	{2}	
3	c	False	{3}	{3}	$\{3\} \cup \{4\}$
C	+	False	{3}	{3}	
D		False	{1,2,3}	{2,3}	
4	#	False	{4}	{4}	
E	•	False	{1,2,3}	{4}	

The final step is to construct the dfa using the firstpos(), lastpos() and followpos() information of Table 3.

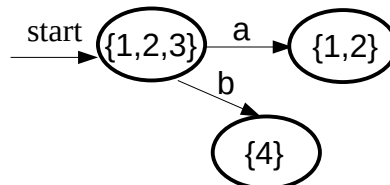
- The start state of the dfa is given by firstpos(root) which is $\{1,2,3\}$ for our regex.



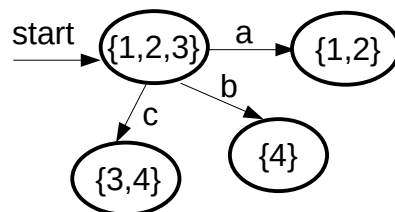
- The state transitions on symbols a, b and c are now calculated.
- For transition on 'a' from the start state, we need to find all positions that correspond to 'a' in $\{1,2,3\}$. The position 1 is the only one that has label 'a'. The next state of $\{1,2,3\}$ on symbol 'a' is given by followpos(1) = $\{1,2\}$.



- For transition on 'b' from the start state, we need to find all positions that correspond to 'b' in $\{1,2,3\}$. The position 2 is the only one that has label 'b'. The next state of $\{1,2,3\}$ on symbol 'b' is given by followpos(2) = $\{4\}$. The dfa at this stage is

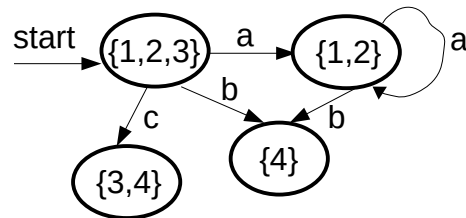


- For transition on 'c' from the start state, we need to find all positions that correspond to 'c' in $\{1,2,3\}$. The position 3 is the only one that has label 'c'. The next state of $\{1,2,3\}$ on symbol 'c' is given by followpos(3) = $\{3, 4\}$. The dfa at this stage is

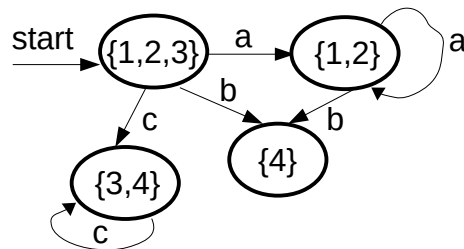


- The state with $\{1,2\}$ is now picked up for possible transitions. On symbol 'a', the next state is given by followpos(1) = $\{1,2\}$, that is the same state. On 'b', the next state is followpos(2) = $\{4\}$. There is no transition from $\{1,2\}$ on symbol 'c'.

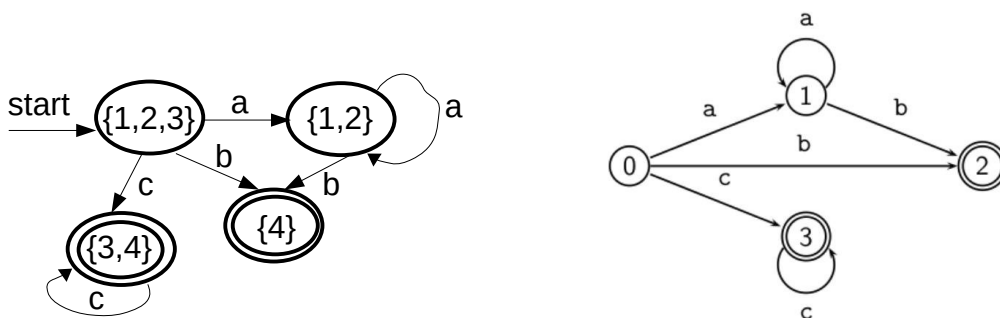
- Adding these transitions produces the following dfa.



- State corresponding to $\{4\}$ has no transitions as 4 does not correspond to any of the alphabet symbols $\{a, b, c\}$.
- The state with $\{3,4\}$ is now picked up for exploration. The only symbol is 'c' that corresponds to position 3. The next state of $\{3, 4\}$ on 'c' is given by $\text{followpos}(3) = \{3,4\}$, that is the same state. The resulting dfa at this stage is



- The dfa is now complete. The final states of the dfa are the states that contain the endmarker symbol #, which has label 4. Hence there are 2 final states, namely $\{4\}$ and $\{3,4\}$. Marking the final states we get the dfa shown below.



Comparing with the dfa given earlier, we can see that the two dfas are identical, except for the labelling of the states.

The direct dfa construction from a regex is summarized in the following algorithm.

Algorithm

1. Construct the tree for $r\#$ for the given regular expression r
2. Construct functions *nullable()*, *firstpos()*, *lastpos()* and *followpos()*
3. Let *firstpos(root)* be the start state. Push it on top of a stack.
While (stack not empty)
do begin
 pop the top state U off the stack; mark it;
 for each input symbol a do
 begin
 let $p_1, p_2, p_3, \dots, p_k$ be the positions in U corresponding to symbol a ;
 Let $V = \text{followpos}(p_1) \cup \text{followpos}(p_2) \cup \dots \cup \text{followpos}(p_k)$;
 place V on stack if not marked and not already in stack;
 make a transition from U to V labeled a ;
 end
end
4. Final states are the states containing the positions corresponding to $\#$.

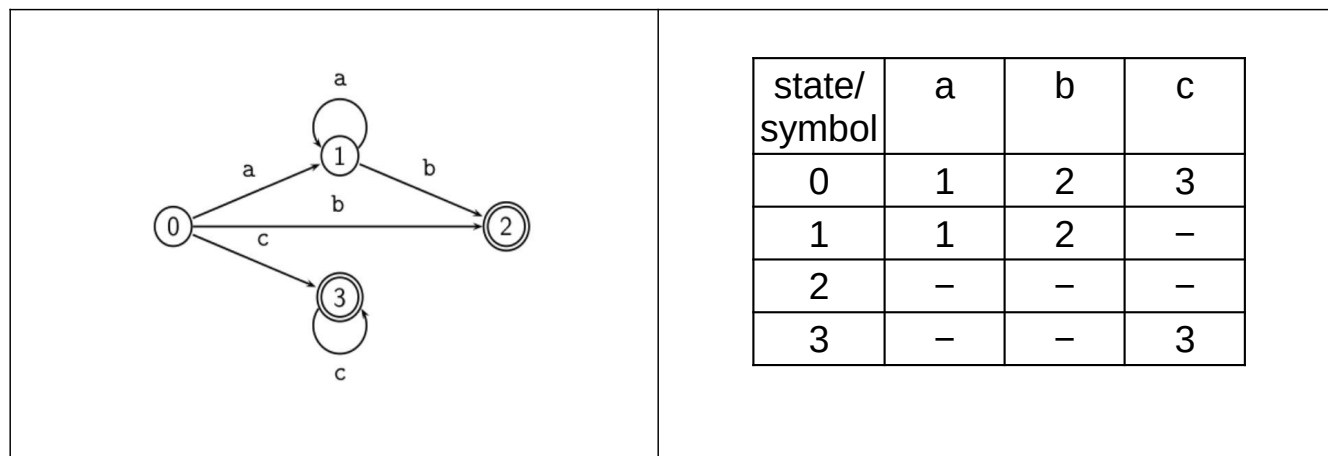
The DFA that is constructed directly from regex is not guaranteed to be the minimal dfa. DFA minimization algorithm is applied to minimize the dfa constructed by the above algorithm.

Representation of a DFA

The dfa that results from the tokens of most programming languages are rather large in size. Therefore a 2-dimensional representation of the dfa is memory intensive, though this representation permits the determination of $\text{nextstate}[s, c]$, where s is a state of the dfa and c is a symbol of the alphabet in constant time, $O(1)$. The dfa for lexical analysers is also fairly sparse leading to wasteful memory space.

The challenge before designers of scanner generation was to find a representation that was memory efficient while retaining the constant access time of $\text{nextstate}[s, c]$.

This issue is explained with the dfa constructed directly from the regex $a^*b \mid c^+$
 An obvious 2-dimensional representation of the dfa is shown.



The 2D representation needs 12 entries as outlined above.

Incomplete : Example for 4-array scheme to be inserted

Illustration of 4 Array Scheme for representing a DFA

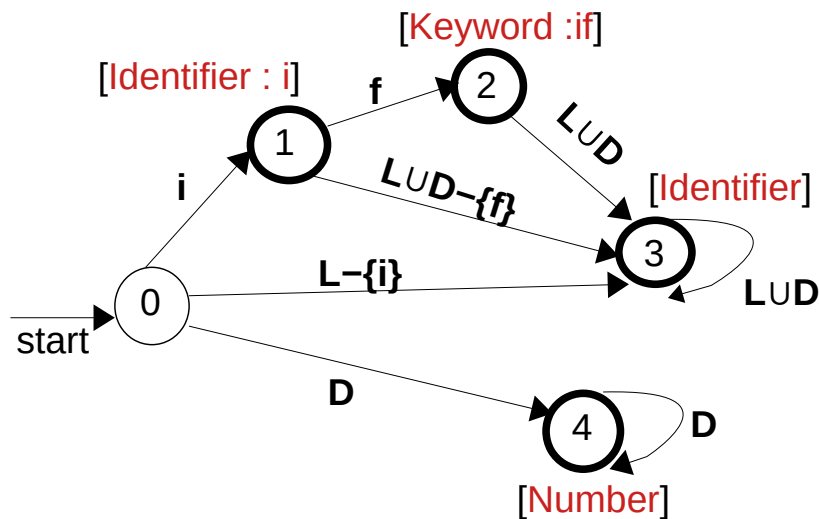
4 array layout of a DFA is the internal representation used by the lexical analyzer generation software, Lex or Flex. The efficacy of this interesting data structure is explained with the help of an example. Consider a lexical analyser for tokens encountered in programming languages. The following 3 tokens, {Keyword, Identifier and Number} are specified and the alphabet is restricted to 15 characters, {a, b, c, d, e, f, g, h, i, j, 0, 1, 2, 3, 4} to contain the size of the dfa that recognizes all the tokens. The symbols from the alphabet are shown in bold. Note that the token Number may admit numbers like {00, 000, ...} but are irrelevant for the present discussion.

Letter	a b c d e f g h i j
Digit	0 1 2 3 4
Keyword	if
Identifier	letter (letter digit)*
Number	digit+

Manually constructed DFA is given below. It is left to the reader to determine the difference of this dfa with respect to the one constructed by the direct regular

expression to DFA algorithm. We use L and D be the abbreviations instead of the expressions Letter and Digit respectively. Note that L has 10 symbols from alphabet while D has 5 numeric symbols. The final states are nodes with bold boundary. The edge labels denote the set of symbols for which transition along the edge happens. The edge labels use sets to indicate the collection of symbols. For example,

L- {i} is a shorthand for the collection, {a, b, c, d, e, f, g, h, j}. The other edge labels are to be elaborated analogously.



The tokens that are recognized at the 4 final states are attached with them in red font.

The symbols of the alphabet are assigned the following offset values :

Table 4 : Symbols and offset values

offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Symbol	a	b	c	d	e	f	g	h	i	j	0	1	2	3	4

1. The 2D representation of this DFA is shown in Table 5.

Table 5 : DFA : 2-Dimensional Representation

State	15 Alphabet Symbols by their offset value														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	3	3	3	3	3	3	3	3	1	3	4	4	4	4	4
1	3	3	3	3	3	2	3	3	3	3	3	3	3	3	3
2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4	–	–	–	–	–	–	–	–	–	–	4	4	4	4	4

Observations from the DFA of Table 5

- The '–' indicates there is no transition from state 4 for all the 10 alphabet characters, 'a' through 'j'
- The space required is for $5 \times 15 = 75$ entries
- All the 15 entries for states 2 and 3 are identical
- 14 entries of state 1 is identical to that of state 2 (and 3); only one entry, [1,5] is different than that of [2, 5] (or [3,5])
- State 0 has a considerable overlap with that of state 2 (and 3); 9 entries are identical while 6 are different.

The C Alphabet, that is the full character set of C, has at least 96 symbols, Alphabet(52 characters), digits (10), and 34 symbols ! " # % & ' () * + , - . / : ; < = > ? [\] ^ _ { | } ~ space horizontal-tab vertical-tab form-feed, newline

Assuming 100 states in the complete DFA, the space requirement is for 10000 entries. However the full table is usually sparse, so there is inefficient space utilization, if the 2D data structure is directly used. The advantage is that the `nextstate[s, offset]`, for any state `s` and any offset (in the range[0, 95]), is obtained in $O(1)$ time.

Because of the size and cost of RAM, during the period when Lex was under development, researchers were looking for a data structure that had the following characteristics :

- Determination of a state transition, `nextstate[s, offset]`, should be efficient, possibly $O(1)$ in practice. This rules out a linked list data structure as a possible candidate, even though it is well known that the sparsity in the dfa can be exploited by a list data structure.

- The overlap between 2 and more states (found to occur very frequently in such dfa), should be exploited to eliminate redundant entries.
- The sparsity present in the DFA be possibly reduced by plugging the holes with useful data.

The data structure employed by the designers of lex to represent the DFA is a novel one and squarely meets the requirements.

Proposed Data Structure using 4 one-dimensional arrays

Four arrays named as Default[], Base[], Next[] and Check[] are used.

The 2 arrays, Default and Base are of the same size as the number of states, $|S|$. The size of the remaining 2 arrays, say m , where $m > |S|$. The challenge is to keep m as low as possible. The function of the arrays, in brief, are described in the following.

- The value of Base[k], for the state k , $0 \leq k \leq |S|-1$, is used as an index in the Next[] and Check[] arrays. Let n be the number of alphabet symbols, $|\Sigma| = n$, with assigned offset indices from 0 to $n-1$.
- Then the n state transitions of state k are laid out in the array positions, Next[Base[k] + 0] to Next[Base[k] + $n-1$].
- The value of Check[j], $0 \leq j \leq n-1$, denote the state for which Next[j] is the destination state.
- Default[k], $0 \leq k \leq |S|-1$, denotes a state that is to be used as a default for state k . A default state, Default[k] shares many transitions with the original state k .

The construction process is better appreciated by working it out for the dfa represented by Table 5.

There are several choices for choosing the layout and in general finding an optimal layout is a NP-hard problem. Heuristics are used to find a good layout, that is an appropriate contiguous segment in the Next[] and Check[] arrays.

1. States 1, 2 and 3 have a large overlap. We choose state 2 to start the layout. WLOG we choose Base[2] to be 0.
 - $\text{nextstate}[2, i] = 3$ for $0 \leq i \leq 14$; therefore $\text{Next}[\text{Base}[2] + i] = 3$; $0 \leq i \leq 14$. Also since all these transitions are for state 2, we assign $\text{Check}[\text{Base}[2] + i] = 2$; $0 \leq i \leq 14$; Since there is no state that 2 shares its entry with (being the first chosen state), we set $\text{Default}[2]$ to be Null (denoted by $-$)
 - The entries of state 2 is now complete and shown in Table 6.
 - You should verify that all the transitions of state 2 are obtained by if ($\text{CHECK}[\text{BASE}[2] + i] == 2$) then return($\text{NEXT}[\text{BASE}[2] + i]$) for all i in $0 \leq i \leq 14$; essentially 3 array references.

Table 6 : 4-arrays after placing transitions of state 2

Index	Default	Base		Index	Next	Check
0				0	3	2
1				1	3	2
2	-	0		2	3	2
3				3	3	2
4				4	3	2
				5	3	2
				6	3	2
				7	3	2
				8	3	2
				9	3	2
				10	3	2
				11	3	2
				12	3	2
				13	3	2
				14	3	2

2. We now choose state 3 for laying out its transitions. Since state 3 is identical to state 2 (already placed), we choose Base[3] as 0 and assign Default[3] to 2.

The transitions of state 3 are now determined by the following logic.

```
if (CHECK[BASE[3] + i] != 3)
    then return (nextstate( DEFAULT[3], i))
```

Interesting to observe that only Base[3] and Default[3] changed while Next[] and Check[] remains unchanged.

Q. How many array references are required to find nextstate[3, i];
 $0 \leq i \leq 14$?

Table 7 : 4-arrays after placing transitions of states 2 and 3

Index	Default	Base		Index	Next	Check
0				0	3	2
1				1	3	2
2	–	0		2	3	2
3	2	0		3	3	2
4				4	3	2
				5	3	2
				6	3	2
				7	3	2
				8	3	2
				9	3	2
				10	3	2
				11	3	2
				12	3	2
				13	3	2
				14	3	2

- The state to be placed now is 0. nextstate[0, 8] is 1 which is different from nextstate[2, 8], but nextstate[0, i] = nextstate[2,i]; $0 \leq i \leq 7$. The setting of Default[0] = 2 will take care of the first 8 transitions of 0. We need a free slot to place nextstate[0, 8] and the first free slot is available at index 15.

To do this, Base[0] must be 7. The nextstate[0, 9] can be served by the default state 2, provided index 16, which base value 7 plus 9 is kept vacant (set to -1). The indices 17 to 21 can be used to save nextstate[0, k]; $10 \leq k \leq 14$. The 4-arrays after placing state 0 is shown in Table 8.

Table 8 : 4-arrays after placing states 2, 3 and 0

Index	Default	Base		Index	Next	Check
0	2	7		0	3	2
1				1	3	2
2	–	0		2	3	2
3	2	0		3	3	2
4				4	3	2
				5	3	2
				6	3	2
				7	3	2
				8	3	2
				9	3	2
				10	3	2
				11	3	2
				12	3	2
				13	3	2
				14	3	2
				15	1	0
				16		-1
				17	4	0
				18	4	0
				19	4	0
				20	4	0
				21	4	0

- State 1 is now chosen for placement. Except for nextstate[1, 5] = 2, all other entries are same as state 2. State 2 should therefore be used as the default state for 1. To place [1, 5], we use the free slot at index 16 by

setting Base[1] to 11. The indices that state 1 will use are from 11 to 25. By setting Check[22] = Check[23] = Check[24] = Check[25] = -1, we ensure that default state 2 will be used for next state information.

Table 9 : 4-arrays after placing states 2, 3, 0 and 1

Index	Default	Base	Index	Next	Check
0	2	7	0	3	2
1	2	11	1	3	2
2	-	0	2	3	2
3	2	0	3	3	2
4			4	3	2
			5	3	2
			6	3	2
			7	3	2
			8	3	2
			9	3	2
			10	3	2
			11	3	2
			12	3	2
			13	3	2
			14	3	2
			15	1	0
			16	2	1
			17	4	0
			18	4	0
			19	4	0
			20	4	0
			21	4	0
			22		-1
			23		-1
			24		-1
			25		-1

5. Finally we have to place state 4. State 4 has 5 common transitions with state 0 but that is not easy to reuse (Why?). So we use the indices 22 to 26 to place the transitions for this state by Base[4] set to 12 with no default state.

Table 10 : 4-arrays after placing all 5 states

Index	Default	Base		Index	Next	Check
0	2	7		0	3	2
1	2	11		1	3	2
2	–	0		2	3	2
3	2	0		3	3	2
4	–	12		4	3	2
				5	3	2
				6	3	2
				7	3	2
				8	3	2
				9	3	2
				10	3	2
				11	3	2
				12	3	2
				13	3	2
				14	3	2
				15	1	0
				16	2	1
				17	4	0
				18	4	0
				19	4	0
				20	4	0
				21	4	0
				22	4	4
				23	4	4
				24	4	4
				25	4	4

Index	Default	Base	Index	Next	Check
			26	4	4

The construction of the 4-array representation for the given DFA is complete. Verify that all the state transition information is identical in both the representations.

The benefits of the scheme for this example may be summarized as

- 4-array scheme uses $26*2 + 5*2 = 62$ units as compared to 75 units for a dense matrix representation
- There is no hole in the Next[] and Check[] arrays so we can claim that dense packing has been achieved. However it is not obvious whether the size of these 2 arrays could be reduced further by better exploitation of common entries.
- While the cost of `nextstate[s, i]` for any `i` is $O(1)$, the 4-array scheme has a higher time complexity. What is the largest number of array references required for `nextstate[s, i]` for any `i`?

This completes the layout of a DFA in the 4-array scheme. The calculation of the next state information in a 4array scheme is outlined in the following function.

```
function nextstate (s, a)
{ if CHECK[BASE[s] + a] = s
  then NEXT[BASE[s] + a]
  else
    return (nextstate( DEFAULT[s], a))
}
```

A heuristic, which works well in practice to fill up the four arrays, is to find for a given state, the lowest BASE, so that the special entries of the state can be filled without conflicting with the existing entries.

Take-away from Lexical Analyser Generators

- A different algorithm, developed by compiler designers, for constructing a DFA directly from a regular expression. The algorithm is efficient and provably correct. The lexical analyzer generator tool Lex uses this algorithm.
- A novel data structure named as 4 array scheme has been designed to store a large DFA efficiently (reduced space requirement) with marginal increase in the cost for finding the nextstate. Lex uses the 4 array scheme as its internal representation for a DFA.
- The direct DFA construction algorithm does not claim that the generated dfa is minimal. Therefore the generated dfa requires a minimization algorithm to produce the minimal dfa.
- For the sake of completeness, an algorithm for state minimization (covered in FLAT) is reproduced below, without explanation, for your perusal.

Minimizing states of a dfa

Outline of an algorithm that minimizes the states S of a given dfa. This has been covered in the course on FLAT and hence details are omitted.

1. Construct an initial partition $\Pi = \{ S - F, F_1, F_2, \dots, F_n \}$, where $F = F_1 \cup F_2 \cup \dots \cup F_n$ and each F_i is the a final state.
2. For each set G in Π do
 - Partition G into subsets such that two states s and t of G are in the same subset if and only if
For all input symbols a , and for states s and t such that
states s and t have transitions onto states in the same set of Π ;
construct a new partition Π_{new} by replacing G in Π by the set of its subsets.
3. If $\Pi_{\text{new}} == \Pi$, then $\Pi_{\text{final}} := \Pi$ and continue with step 4.
Otherwise repeat step 2 with $\Pi := \Pi_{\text{new}}$.
4. Merge states in the same set of the partition.

5. Remove any dead (unreachable) states.

Reporting of typos / errors in this document will be appreciated.

End of Notes on Lexical Analysis

Principles of Top-Down Parsing

A top down parser creates a parse tree starting with the root, i. e., it starts a derivation from the start symbol of the grammar.

1. It could potentially replace any of the nonterminals in the current sentential form. However while tracing out a derivation , the goal is to produce the input string.
2. Since the input tokens are scanned from left to right , it will be desirable to construct a derivation that also produces terminals in the left to right fashion in the sentential forms.
3. A leftmost derivation matches the requirement exactly. A property of such a derivation is that there are only terminal symbols preceding the leftmost nonterminal in any leftmost sentential form.
4. The basic step in a top down parser is to find a candidate production rule (the one that could potentially produce a match for the input symbol under examination) in order to move from a left most sentential form to its succeeding left sentential form.
5. A top down parser uses a production rule in the obvious way - replace the lhs of the rule by one of its rhs.


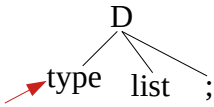
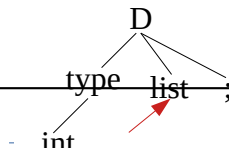
Example to Illustrate the Principles of Top-Down Parsing

Top-down parsing of the sentence “**int a, b;**” for the C declaration grammar, named as G1, given below.

R1 : $D \rightarrow \text{type list ;}$ R2 : $\text{type} \rightarrow \text{int}$
 R3 : $\text{type} \rightarrow \text{float}$ R4 : $\text{list} \rightarrow \text{id L}$
 R5 : $L \rightarrow , \text{id L}$ R6 : $L \rightarrow \epsilon$

The start symbol of the CFG is the nonterminal D. The rules have been numbered for ease of reference.

The processing of a top down parser over the given CFG and the input can be understood by tracing the moves of the parser. The current token in the input is highlighted and an arrow points to the leftmost terminal in the sentential form.

Input	Rule used with Explanation	Incremental Generation of Parse Tree
int a , b ;	None	Initial Parse Tree 
int a , b ;	Leftmost nonterminal is D R1 : $D \rightarrow \text{type list ;}$	
int a , b ;	Leftmost nonterminal is type R2 : $\text{type} \rightarrow \text{int}$ int matches with the current token	

Input	Rule used with Explanation	Incremental Generation of Parse Tree
	get next token; mark leftmost nonterminal	
int a , b ;	Leftmost nonterminal is list R4 : list \rightarrow id L a matches with current token id get next token; mark leftmost nonterminal	
int a , b ;	Leftmost nonterminal is L R5 : L \rightarrow , list , matches with current token , get next token; mark leftmost nonterminal	
int a , b , ;	Leftmost nonterminal is list R4 : list \rightarrow id L b matches with current token id get next token; mark leftmost nonterminal	
int a , b ; ,	Leftmost nonterminal is L R6 : L \rightarrow ϵ There are no leftmost nonterminal left in the parse tree The nexttoken ; matches with the ';' of rule R1	

At this stage the entire input string is exhausted and has been successfully expanded from the start symbol D. Not only the parsing has been successful, the process has also traced out a leftmost derivation.

Let us recap the production rules used for parsing. The nonterminal that has been expanded in a sentential form has been highlighted and it clearly shows that a leftmost derivation has been traced out while parsing the input sentence.

$D \Rightarrow \text{type list ;} \Rightarrow \text{int list ;} \Rightarrow \text{int id L ;} \Rightarrow \text{int id , id L ;} \Rightarrow \text{int id , id ;}$

The yield of the parse tree matches exactly the input string.

Q. How would Top Down parsing proceed if we had instead used the following equivalent grammar, say G2, given below ?

R1 : $D \rightarrow \text{type list ;}$
R2 : $\text{type} \rightarrow \text{int}$
R3 : $\text{type} \rightarrow \text{float}$
R4 : $\text{list} \rightarrow \text{list , id}$
R5 : $\text{list} \rightarrow \text{id}$

Analysing the CFG to extract Useful Information

Processing the production rules of CFG generates several useful information about its nonterminals. Let us consider the grammar G1.

R1 : $D \rightarrow \text{type list ;}$
R2 : $\text{type} \rightarrow \text{int}$
R3 : $\text{type} \rightarrow \text{float}$
R4 : $\text{list} \rightarrow \text{id L}$
R5 : $L \rightarrow \text{ , id list}$
R6 : $L \rightarrow \epsilon$

- A nonterminal including the start nonterminal can derive several sentential forms using the grammar rules. The nonterminals are {D, type, list, L}
- Consider the following derivations from D

$D \Rightarrow \text{type list ;} \Rightarrow \text{int list ;} \quad D \Rightarrow \text{type list ;} \Rightarrow \text{float list ;}$

While D can derive many more sentential forms, it can be seen that the first terminals that can appear in any sentential form (leftmost, rightmost or an arbitrary one) that are derivable from D are just two, namely {int, float}.

Similar information about the other non-terminals of G1 can also be obtained. This information about a nonterminal, say A, is denoted by FIRST(A), and is very useful in parsing.

For example, when the first token of an input, say “short”, is checked for syntactic validity with respect to G1, a top down parser will try to match “short” with $\text{FIRST}(D) = \{\text{int, float}\}$ and since this match fails, the parser will declare an error.

In general, it may appear at a first glance that for a given G, to construct FIRST(A), for all A in G, one would have to generate all the sentential forms possibly in G. Fortunately that is not the case and these sets can

be easily constructed using an algorithm that processes the rules iteratively and gets the desired information within a few iterations.

Construction of FIRST() sets

The set of terminals that can ever appear in any sentential form derivable from a string α , $\alpha \in (N \cup T)^*$, is denoted by $FIRST(\alpha)$.

Formally, this is defined as $FIRST(\alpha) = \{ a \in T^* \mid \alpha \Rightarrow^* a \beta \text{ for some } \beta \}$.

It follows that if $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in FIRST(\alpha)$

Some obvious facts about FIRST information are the following :

1. For a terminal $a \in T$, $FIRST(a) = \{a\}$
2. For a nonterminal A , if $A \rightarrow \epsilon$, then $\epsilon \in FIRST(A)$
3. For a rule $A \rightarrow X_1 X_2 \dots X_r$

$$FIRST(A) = FIRST(A) \cup_k (FIRST(X_k)); 1 \leq k \leq r; \epsilon \in X_i; 1 \leq i \leq k-1;$$

The expression above says that the contributions from the FIRST sets of X_2, X_3, \dots, X_{k-1} also need to be computed if all their preceding nonterminals can derive ϵ . For instance, the terminals in $FIRST(X_5)$ are to be added to $FIRST(A)$, only if, $\epsilon \in FIRST(X_1)$ and $\epsilon \in FIRST(X_2)$ and $\epsilon \in FIRST(X_3)$ and $\epsilon \in FIRST(X_4)$.

An Algorithm that computes FIRST sets of nonterminals for a general CFG follows.

Algorithm for FIRST sets

Input : Grammar $G = (N, T, P, S)$

Output : $FIRST(A)$, $A \in N$

Method :

begin algorithm

for $A \in N$ do $FIRST(A) = \Phi$;

 change := true;

 while change do

 { change := false;

 for $p \in P$ such that p is $A \rightarrow \alpha$ do

 { newFIRST(A) := $FIRST(A) \cup Y$; where Y is $FIRST(\alpha)$ from definition given earlier;

 if newFIRST(A) \neq FIRST(A) then

 { change := true ; FIRST(A) := newFIRST(A); }

 }

end algorithm

Illustration of FIRST() Set Computation :

R1 : $D \rightarrow \text{type list}$;	R2 : $\text{type} \rightarrow \text{int}$
R3 : $\text{type} \rightarrow \text{float}$	R4 : $\text{list} \rightarrow \text{id } L$
R5 : $L \rightarrow , \text{list}$	R6 : $L \rightarrow \epsilon$

Nonterminal	Initialization	Iteration 1	Iteration 2	Iteration 3
D	Φ	Φ	{int float}	{int float}
type	Φ	{int float}	{int float}	{int float}
list	Φ	{id}	{id}	{id}
L	Φ	{ ϵ ,}	{ ϵ ,}	{ ϵ ,}
change	-	True	True	False

The working of the algorithm can be explained as follows. The initialization step is obvious. In the first iteration, rule R1 is used for $\text{FIRST}(D) = \text{FIRST}(\text{type list ;}) = \text{FIRST}(\text{type}) = \Phi$

$\text{FIRST}(\text{type}) = \text{FIRST}(\text{int}) = \{\text{int}\}$ using R2 and using R3, we get $\text{FIRST}(\text{type}) = \text{FIRST}(\text{type}) \cup \text{FIRST}(\text{float}) = \{\text{int float}\}$. Since $\text{FIRST}(\text{type})$ has changed from Φ to $\{\text{int float}\}$, change is set to True. Similarly in the second iteration, $\text{FIRST}(D)$ changes due to which change is again set to True. All the $\text{FIRST}()$ sets in iteration 3 remain the same as they were at the end of iteration 2, which causes change to be assigned False and the algorithm terminates with the $\text{FIRST}()$ sets as given at the end of iteration 3.

Q. Can you construct a derivation that generates a sentential form justifying the presence of a terminal in the $\text{FIRST}(A)$ for any A?

Q. What is the worst complexity of the algorithm in terms of the quantities present in the grammar?

Another Useful Information From the CFG

Another information that is found to be very useful in several parsing methods (but not all) is also based on sentential forms that are generated by the grammar rules.

- Consider the following derivations from D using grammar G1

R1 : $D \rightarrow \text{type list ;}$ R2 : $\text{type} \rightarrow \text{int}$
 R3 : $\text{type} \rightarrow \text{float}$ R4 : $\text{list} \rightarrow \text{id L}$
 R5 : $L \rightarrow , \text{list}$ R6 : $L \rightarrow \epsilon$

$D \Rightarrow \text{type list ;} \Rightarrow \text{int list ;}$ $D \Rightarrow \text{type list ;} \Rightarrow \text{type id L ;}$

$D \Rightarrow \text{type list ;} \Rightarrow \text{float list ;}$

$\text{list} \Rightarrow \text{id L} \Rightarrow \text{id , list} \Rightarrow \text{id , id L} \Rightarrow \dots$

- The information that is sought now is about the terminals that appear immediately to the right of a nonterminal in any sentential form of G1. Such information is named as $\text{FOLLOW}(A)$ for a nonterminal A and gives the terminal symbols that immediately follow A in some sentential form of G1.
- For instance, using R1 it can be seen that $\text{FOLLOW}(\text{list})$ contains ‘;’ similarly from the derivation sequence, “ $D \Rightarrow \text{type list ;} \Rightarrow \text{type id L ;}$ ” we find that $\text{FOLLOW}(L)$ contains ‘;’. It is intended to construct $\text{FOLLOW}(A)$ for all A in N.

Rules for construction of FOLLOW() sets are the following.

F1 : For the start symbol of the grammar, $\$ \in \text{FOLLOW}(S)$. The rationale for this special rule is that it is essential to know when the entire input has been examined. A special marker, \$, is appended to be actual input. Since S is the start symbol, only the end of input marker, \$, can follow S.

F2 : If $A \rightarrow \alpha B \beta$ is a grammar rule, then $\text{FOLLOW}(B) \cup \{\text{FIRST}(\beta) - \epsilon\}$. This operator “ \cup ” is used in the same sense as “ $+=$ ” of C, that is the existing terminals in FOLLOW(B), add all the terminals in FIRST(β) to it. Ignore ϵ even if ϵ belongs to FIRST(β).

F3 : If either $A \rightarrow \alpha B$ is a rule or if $A \rightarrow \alpha B \beta$ is a rule and $\beta \Rightarrow^* \epsilon$ (that is, $\epsilon \in \text{FIRST}(\beta)$), then $\text{FOLLOW}(B) \cup \text{FOLLOW}(A)$

The rule says that under the given conditions, whatever follows A also follows B and this fact can be verified by using the give rule in a derivation.

- The rules for FOLLOW() can be applied to write an algorithm that constructs FOLLOW(A) for all A in N iteratively, similar to the algorithm for construction of FIRST sets.

G1 is reproduced below for ready reference.

R1 : $D \rightarrow \text{type list ;}$ R2 : $\text{type} \rightarrow \text{int}$
R3 : $\text{type} \rightarrow \text{float}$ R4 : $\text{list} \rightarrow \text{id L}$
R5 : $L \rightarrow , \text{list}$ R6 : $L \rightarrow \epsilon$

- F1 applied to R1 gives $\text{FOLLOW}(D) = \{\$\}$.
F2 applied to R1 gives $\text{FOLLOW}(\text{type}) \cup \text{FIRST}(\text{list}) = \{\text{id}\}$;
 $\text{FOLLOW}(\text{list}) \cup \text{FIRST}\{;\} = \{;\}$
- None of F1, F2 or F3 apply to R2 and R3.
- F3 applied to R4 gives $\text{FOLLOW}(L) \cup \text{FOLLOW}(\text{list}) = \{;\}$
- F3 applied to R5 gives $\text{FOLLOW}(\text{list}) \cup \text{FOLLOW}(L) = \{;\}$
- Processing of the FOLLOW() sets iteratively shows that the information converges at the end of iteration 2.

A in N	FIRST()	Initialization	Iteration 1	Iteration 2
D	{int float}	{\$}	{\$}	{\$}
type	{int float}	Φ	{id}	{id}
list	{id}	Φ	{;}	{;}
L	{ ϵ ,}	Φ	{;}	{;}
change		NA	True	False

Q. Can you construct a derivation that generates a sentential form justifying the presence of a terminal in the FOLLOW(A) for any A.?

Q. Write an algorithm for construction of FOLLOW() for all nonterminal symbols of a CFG, on the same lines as given for FIRST() for all nonterminals.

Q. What is the worst complexity of your algorithm in terms of the quantities present in the grammar?

TOP-DOWN PARSER CONSTRUCTION

Deterministic Top-Down Parser

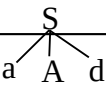
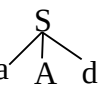
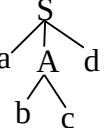
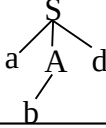
Consider the grammar given below with S as the start symbol.

1. $S \rightarrow aAd$
- 2, 3) $A \rightarrow bc \mid b$

Parse the input **a c d** using a parser that is permitted to backtrack its moves, that is, in the case that it has made a wrong move and is not able to match the input, it can retract some move and try again. To make the backtracking deterministic, we shall try to backtrack the last move and start afresh. The objective is not to recommend backtracking parsers for implementation, but to reason why deterministic parsers, that never go back on its moves to parse an input, are so important for practical applications.

The starting point : Start nonterminal S and the input a c d

Parser chooses rule 1 as its move. The result is that $S \Rightarrow a A d$ and i/p : **a** b d; highlighted symbols match, new configuration is : nonterminal A now has to match the remaining input: b d. The moves of a backtracking parser for this grammar and input are summarized in the table below.

Sentential Form	Input	Rule and Derivation	Parse Tree
S	a c d	Use the $S \Rightarrow a A d$	
$\Rightarrow a A d$	a c d	token a matches; get next token and use the leftmost nonterminal	
$\Rightarrow a A d$	a c d	Try one of the rule for A, $A \rightarrow bc$	
$\Rightarrow a b c d$	a c d	b does not match with c	Parse fails : backtrack the last rule used
$\Rightarrow a A d$	a c d	Try the other rule for A, 3) $A \rightarrow b$	

Sentential Form	Input	Rule and Derivation	Parse Tree
$\Rightarrow a b d$	$a c d$	b does not match with c; Parse fails : backtrack the last rule; both rules for A fail; backtrack again.	<pre> graph TD S --> a S --> A S --> d </pre>
$S \Rightarrow a A d$	$a c d$	Try another rule for S but there is no other rule for S	S
S	$a c d$	No rule is left unexplored at this point.	Parse fails as there is no move left unexplored; hence "acd" does not belong to G

To compensate for no backtracking, some additional information is made available to a deterministic parser.

1. For instance, a token the parser is trying to match ; the current token being scanned in the input is known as the lookahead symbol.
2. Knowing the lookahead symbol helps the parser to make the selection. For example, let **a** be the lookahead symbol, and A be the leftmost nonterminal (lm nonterminal) in a leftmost sentential form. Let the rules for A be
 $A \rightarrow a\alpha \mid b\beta$

It should be clear from the two rules for A, that the former rule, $A \rightarrow a\alpha$ is the right choice in this situation, since the other fails to match the lookahead token. In general, for a nonterminal A, the rule for A which contains the lookahead token as the first symbol in the rhs is a correct choice.

3. It is not necessary that for a nonterminal to have a terminal as the first symbol. How does a parser choose a rule in such a situation? Let the rhs of a rule, corresponding to a lm nonterminal, say A, have a nonterminal as the first symbol, such as $A \rightarrow C\gamma \mid B\beta$ which have nonterminals C and B respectively in the rhs of all the rules for A.

4. Given the lookahead symbol **t** to be matched, the rule that should be used to match **t** can not be determined directly from the rhs of the rules, $A \rightarrow C\gamma \mid B\beta$. If one could determine whether C (or B) derives a string whose first symbol is the lookahead, that is, whether $t \in \text{FIRST}(C\gamma)$ or $t \in \text{FIRST}(B\beta)$ then the choice is again possible, provided t belongs to one of two FIRST() sets. In the case that $t \in \{\text{FIRST}(C\gamma) \cap \text{FIRST}(B\beta)\}$, then again there is a problem because both the rules can be applied for parsing and the grammar is then ambiguous for the top down parser.

Basic Steps of a top-down parser

1. Such a parser uses a rule by using its rhs a symbol at a time. A terminal symbol in the rhs must match the lookahead, else an error is reported.
2. A nonterminal symbol in the rhs calls for its expansion by choosing a suitable alternate from the rules associated with the lhs nonterminal.
3. Successful parse is indicated when the parser is able to consume all the tokens in the input. To ease the detection of end of input, a special symbol, \$, is used. Otherwise an error is indicated at the point where the match failed to occur.

Left Recursive Grammar and Top-Down Parsing

Left recursive grammars could cause a top down parser to get into an infinite loop, even if the grammar is unambiguous and the input happens to be a valid sentence. The following example illustrates this fact.

Example : Consider the rule $E \rightarrow E + id \mid id$ and the input $id + id + id \$$. The grammar is left recursive because of the first rule.

- The parser could use the rule $E \rightarrow id$, in which case the lookahead symbol being id , a match is found. The parser reports an error subsequently, since the next lookahead, $+$, does not match the rest of the rhs.
- If it uses the other rule, then the parser goes into an infinite loop. This is because if the parser move is to use the rule $E \rightarrow E + id$, then after the expansion, the lm nonterminal is still E and the lookahead is id , so the same rule has to be applied again. Note that even if we had a backtracking top down parser, for this grammar the problem still remains.
- The conclusion is that left recursive are not to used with top down parsing.

We have already seen how left recursion can be removed from a CFG without changing the underlying language. Consider manual construction of a top down parser. It is assumed that the CFG is available and is free from left recursion.

- Since the rhs of the rules guide such a parser, a possible approach is to convert the rules directly into a program.
 - For each nonterminal a separate procedure is written.
 - The body of the procedure is essentially the rhs of the rules associated with the nonterminal. The basic steps of construction are as follows.
1. For each alternate of a rule, the rhs is converted into code symbol by symbol, as explained below.
 - If a symbol is a terminal, it is matched with the lookahead. The movement of the lookahead symbol on a match can be done by writing a separate procedure for it.
 - If the symbol is a nonterminal, call to the procedure corresponding to this nonterminal is made in its place.
 2. Code for the different alternates of this nonterminal are appropriately combined to complete the body of the procedure.
 3. The parser is activated by invoking the procedure corresponding to the start symbol of the grammar.

The process is illustrated through an example.

Example : Consider the rules given below :

$exp \rightarrow id\ exprime$
 $exprime \rightarrow + id\ exprime \mid \epsilon$

- For the first rule, $exp \rightarrow id\ exprime$, we write a procedure named **exp** as follows. As we can observe, the rhs of exp is directly coded as the body of exp .

```

procedure exp()
begin
    if lookahead = id then
        begin
            match (id) ; exprime()      // same as rhs of exp written in code form
        end
    else error
    if lookahead = $ then report success // equivalent to  $\$ \in \text{FOLLOW}(\text{exp})$ 
    else error
end ;

```

- The code for procedure match is now written. The match() procedure in the event of a match gets the next token, else it returns error because the match fails.

```

procedure match(token t);
begin
    if lookahead = t then
        lookahead := nexttoken ;
    else error
end ;

```

- The body of the final procedure for exprime() is written similarly.

```

procedure exprime ()
begin
    if lookahead = + then
        begin
            match (+) ;
            if lookahead = id then
                begin
                    match ( id ) ; exprime ()
                end
            else error
            end
    else null ;
end ;

```

Recursive Descent Parser

Non-backtracking form of a top-down parser is also known as a predictive parser. The parser constructed manually above is a predictive parser.

A parser that uses a collection of recursive procedures for parsing its input is called a recursive descent parser. The procedures may be recursive (because of rules containing recursion). We constructed a recursive descent parser for the example grammar above.

Exercise : Modify the code for the procedures exp and exprime so that the rules that are used while parsing are also printed.

Comments on Recursive Descent Parsers

1. A recursive descent parser is easy to construct manually. However an essential requirement is that the language in which the parser is being written must support recursion.
2. Certain internal details of parsing are not directly accessible, for example
 - (a) the current leftmost sentential form that this parser is constructing,
 - (b) the stack containing the recursive calls active at any instant is not available for inspection or manipulation.
3. If there is a rule $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$, that is more than one alternates beginning with the same symbol in the rhs, then writing the code for procedure A is nontrivial. The method used in the example would fail to work for such cases.
4. A solution to the problem mentioned above is called **left factoring**. A rule such as

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$$

where γ does not begin with α , can be equivalently written as

$$\begin{aligned} A &\rightarrow \alpha A_0 \mid \gamma \\ A_0 &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{aligned}$$

5. A crucial question is whether one can always write a recursive descent parser (RD parser) for a context free grammar. The answer is no. The construction process, however, does not provide much help in characterizing the subclass of CFGs that admit a RD parser.

Table Driven Top Down Parser

We now study another top-down parsing algorithm. It is a non-recursive version of the recursive descent parser and happens to be the most popular among the top down parsers. It is commonly known as a LL(1) parser.

Writing an LL(1) Parser

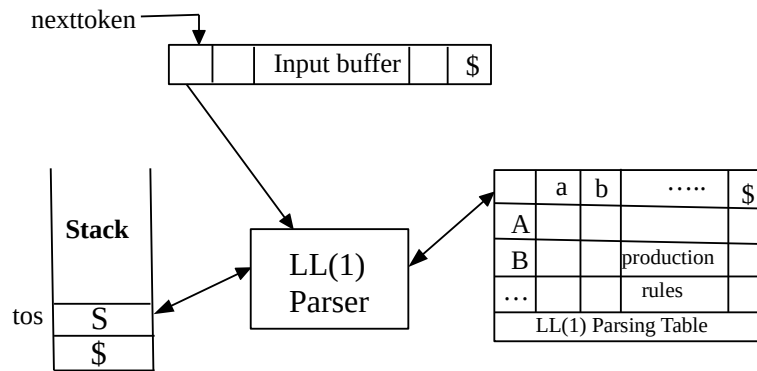
Why the name LL(1) ?

The first L stands for direction of reading the input, L indicates left to right scanning of input; the next L indicates the derivation it uses (leftmost in this case) and 1 is the number of lookahead symbols used by the parser. LL(k), $k > 1$, that is LL parser with k lookahead symbols, is only of theoretical interest because of the increase in the size of parser and also its complexity.

The components of an LL(1) parser is shown in Figure. The data structures employed by this parser are

- a stack
- a parsing table, and
- a lookahead symbol.

1. The purpose of the stack is to hold left sentential forms (or parts thereof). Since the requirement is to expand the leftmost nonterminal, the symbols in the rhs of a rule are pushed into the stack in the reverse order (from right to left).
2. The table has a row for every nonterminal and a column for every terminal (an entry for input-end-marker \$ as well).
3. $TABLE[A,t]$ either contains an error or a production rule.



Initial Configuration of LL(1) Parser

The working of LL(1) parser is given below, which is self explanatory.

tos	nexttoken	Parser actions
\$	\$	Successful parse; Halt
a	a	pop a; get nexttoken; continue
a	b	error
A	a	Table[A, a] = '-' : where - denotes an error
A	a	Table[A, a] = $A \rightarrow XYZ$ pop A; push Z; push Y; push X; continue

Construction of Ll(1) Parsing Table

The parser consults the entry $TABLE[A,t]$ when A is the lm nonterminal and t is the lookahead token. The table encodes all the critical parsing decisions and guides the parser. Such parsers are also known as table driven parsers.

The parsing algorithm is straightforward. The starting configuration is as shown in Figure.

- The top of stack element (tos) along with the lookahead token define a configuration of an LL(1) parser.
- The parser moves from one configuration to another by performing the actions given in the figure.
- The input is successfully parsed if the parser reaches the halting configuration.
- When the stack is empty and nexttoken is \$, it corresponds to successful parse. To simplify detection of empty stack, \$ is pushed at the bottom of the stack.
- Thus $tos = nexttoken = \$$ is the condition for testing the halting configuration.
- The LL(1) Parser is a driver routine which refers to the parsing table, the lookahead token and manipulates the stack.

Illustration of LL(1) Table Construction for Example Grammar

p1 : decls \rightarrow decl decls p2 : decls $\rightarrow \epsilon$
 p3 : decl \rightarrow var list : type ; p4 : list \rightarrow id rlist
 p5 : rlist \rightarrow , id rlist p6 : rlist $\rightarrow \epsilon$
 p7 : type \rightarrow integer p8 : type \rightarrow real

Construction of FIRST() sets

	Init	Iter 1	Iter 2	Iter 3
decls	Φ	$\{\epsilon\}$	$\{\epsilon \text{ var}\}$	$\{\epsilon \text{ var}\}$
decl	Φ	$\{\text{var}\}$	$\{\text{var}\}$	$\{\text{var}\}$
list	Φ	$\{\text{id}\}$	$\{\text{id}\}$	$\{\text{id}\}$
rlist	Φ	$\{, \epsilon\}$	$\{, \epsilon\}$	$\{, \epsilon\}$
type	Φ	$\{\text{integer real}\}$	$\{\text{integer real}\}$	$\{\text{integer real}\}$

Construction of FOLLOW() sets

	Init	Iter 1	Iter 2
decls	Φ	$\{\$\}$	$\{\$\}$
decl	Φ	$\{\text{var}\}$	$\{\text{var}\}$
list	Φ	$\{:\}$	$\{:\}$
rlist	Φ	$\{:\}$	$\{:\}$
type	Φ	$\{;\}$	$\{;\}$

To place the rule, “p1 : decls \rightarrow decl decls”, FIRST(decls) = FIRST(decl decls) = {var}. This results in, TABLE[decls, var] = p1. Also because of “p2 : decls $\rightarrow \epsilon$ ”, using FOLLOW(decls) = { $\$$ }, we place TABLE[decls, $\$$] = p2. The remaining entries of first column is ‘-’ because $\text{var} \notin \{\text{FIRST}(\text{list}) \cap \text{FIRST}(\text{rlist}) \cap \text{FIRST}(\text{type})\}$. Similarly since FIRST(list) = FIRST(id rlist) = {id}, we have the entry TABLE[list, id] = p4. The entries of the LL(1) table can be obtained by using the FIRST() and FOLLOW() sets for the rules. The final parsing table is given below.

	var	id	:	;	,	real	integer	\$
decls	p1	-	-	-	-	-	-	p2
decl	p3	-	-	-	-	-	-	-
list	-	p4	-	-	-	-	-	-
rlist	-	-	p6	-	p5	-	-	-
type	-	-	-	-	-	p8	p7	-

Construction of LL(1) Parsing Table

All that remains is to use the FIRST() and FOLLOW() sets and construct the table row by row of the table.

1. Create a row of the table for each nonterminal of the grammar.
2. The entry for the rule $A \rightarrow \alpha$ is done as follows.
For each $a \in \text{FIRST}(\alpha)$, create an entry $\text{TABLE}[A, a] = \{A \rightarrow \alpha\}$
3. If $\epsilon \in \text{FIRST}(\alpha)$, then create an entry $\text{TABLE}[A, t] = \{A \rightarrow \alpha\}$, where $t \in T \cup \{\$\}$ and $t \in \text{FOLLOW}(A)$.
4. All the remaining entries of the table are marked as “error”.

Remarks on LL(1) Parser

1. This method is capable of providing more details of the internals of the parsing process. For instance, the leftmost sentential form corresponding to any parser configuration can be easily obtained.
2. The syntax error situations are exhaustively and explicitly recorded in the table, LL(1) parser is guaranteed to catch all the errors.
3. How does one know whether a cfg G admits an LL(1) parser ? If the parsing table has unique entries, the resulting parser would work correctly for all sentences of $L(G)$. However, if any entry in the table has multiple rules, the parser would not work and such a grammar is said to be LL(1) ambiguous.
4. The table construction leads to a characterization of the grammar that such a parser can handle. A grammar whose parsing table has no multiply defined entries is called a LL(1) grammar.

Characterization of LL(1) Parser

A characterization of an LL(1) grammar is a fallout of the theory of LL(1) parsing.

Let $A \rightarrow \alpha \mid \beta$ be two distinct production rules in a grammar. Purpose is to find out the conditions under which multiple entries for $\text{TABLE}[A, a]$, for some a , occurs.

- i) $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \{a\}$
- ii) $\epsilon \in \{\text{FIRST}(\alpha) \cap \text{FIRST}(\beta)\}$, multiple entries for all $a \in \text{FOLLOW}(A)$
- iii) $\text{FOLLOW}(A) \cap \text{FIRST}(\beta) = \{a\}$ and $\epsilon \in \text{FIRST}(\alpha)$; then the corresponding table entry has both the rules for A (mutentryz0; a similar situation exists for $\epsilon \in \text{FIRST}(\beta)$.
- iv) there are no other possibilities.

By complementing the conditions (i) to (iii) given above, we get a necessary and sufficient condition for a grammar to be LL(1).

An ambiguous grammar, such as the if-then-else grammar, is also LL(1) ambiguous (see figure below), but the converse is not necessarily true.

LL(1) Table for Dangling Else Grammar

Grammar Rules	p1 : $S \rightarrow i E t S S'$	p2 : $S \rightarrow o$
	p3 : $S' \rightarrow e$	p4 : $S' \rightarrow \epsilon$
	p5 : $E \rightarrow c$	

We have used abbreviations for the terminals : i for **if**; t for **then**; e for **else** for sake of convenience; E denotes an expression, however since it not essential for conditional statements, E is grounded to a terminal c; similarly statements of other language features, such as declarations, expressions, loops, function call, etc. are swept under the carpet and treated as a terminal named o.

Rule p1 sets the entry $TABLE[S, i]$ to p1. Similarly for rule p2, we get $TABLE[S, o]$ to p2. Rule p3 causes $TABLE[S', e]$ to p3. For rule p4, since $S' \rightarrow \epsilon$, only for this rule we need $FOLLOW(S')$. $FOLLOW(S') = FOLLOW(S) = \{e, \$\}$, because of which $TABLE[S', e]$ to p4 and also $TABLE[S', \$]$ to p4. At this point, $TABLE[S', e] = \{p3, p4\}$. Finally rule p5 causes $TABLE[E, c]$ to be set to p5. The table is now complete and it is shown below.

	o	c	e	i	t	\$
S	p2	–		p1		
S'			p3 p4			p4
E		p5				

The given if-then-else grammar is said to be LL(1) ambiguous because when S' is the leftmost nonterminal and the lookahead token is **e**, this table driven parser has 2 valid moves, which is not permitted in a deterministic parser.

LL(1) Parser Generation

The manual construction of LL(1) parser reveals that the entire process can be automated. Figure below shows the internal steps for generating LL(1) parser automatically from the CFG description of a PL.

The first step in the construction process is to transform the given grammar G to an equivalent grammar G' in case the given grammar has left recursion and /or left factoring. Note that the language of both grammars remain the same, that is, $L(G) = L(G')$.

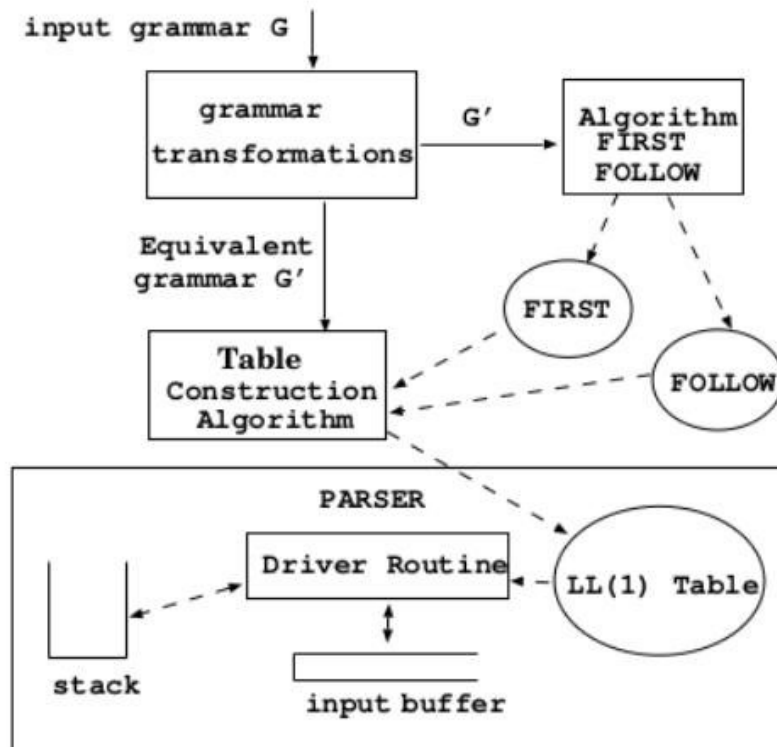
The transformed grammar G' is now processed to construct the $FIRST()$ and $FOLLOW()$ sets for every nonterminals as explained earlier.

The nonterminals of the grammar A , which are not nullable, that is A does not derive ϵ , are processed along with the $FIRST()$ sets to create some entries of the parsing table.

Finally, for the nullable nonterminals, $A \Rightarrow^* \epsilon$, we use $FOLLOW(A)$ to fill the remaining entries of the table. All the remaining entries are marked as error. This completes the construction of the LL(1) parsing table. A generic driver routine, which reads the lookahead token, and takes action based on the symbol (non-terminal or terminal) at the top of stack (tos) is written once for all included.

The driver routine, the parsing stack, along with the parsing table, and an input buffer, constitutes the top down parser. However it must be remembered that the parser is deterministic provided the table has at most one entry in all the positions, else the parser is not deterministic and the grammar is LL(1) ambiguous.

The parser generation process is illustrated in the following figure.



BOTTOM-UP PARSING

Principles Of Bottom Up Parsing

In bottom-up parsing,

1. a parse tree is created from leaves upwards. The symbols in the input, after completion of successful parse, are placed at the leaf nodes of the tree.
2. Starting from the leaves, the parser fills in the internal nodes of the unknown parse tree gradually, eventually ending at the root.
3. The creation of an internal node involves replacing symbols from $(N \cup T)^*$ by a single nonterminal repeatedly. The task is to identify the rhs of a production rule in the tree constructed so far and replace it by the corresponding lhs of the rule. This kind of use of a production rule is called a **reduction**.
4. The crucial task, therefore, is to find the production rules that have to be used for reduction of a sentence to the start symbol of the grammar.

Q. Is there a derivation that corresponds to a bottom-up construction of the parse tree ?

The construction process indicates that it has to trace out some derivation in the reverse order (provided one such exists). The following example provides the answer.

Example : Consider the following declaration grammar from Pascal language, which has 3 nonterminals and the terminals are marked in bold.

$D \rightarrow \text{var list : type ;}$
 $\text{type} \rightarrow \text{integer} \mid \text{real}$
 $\text{list} \rightarrow \text{list , id} \mid \text{id}$

The input string to be parsed is `var id, id : integer ;`

The reductions used and the parse tree construction from the leaves as the input string is processed is shown in Figure below.

Input	Production Rule used	Parse Tree
var id, id : integer ;	None	NULL
id, id : integer ;	None	var
, id : integer ;	list \rightarrow id	<pre>list var id</pre>
id : integer ;	None	<pre>list var id ,</pre>

Input	Production Rule used	Parse Tree
: integer ;	list \rightarrow list , id	<pre> graph TD list1[list] --- list2[list] list1 --- comma1['] list1 --- id1[id] list2 --- var[var] list2 --- id2[id] </pre>
integer ;		<pre> graph TD list1[list] --- list2[list] list1 --- comma1['] list1 --- id1[id] list2 --- var[var] list2 --- id2[id] list1 --- colon1['] list1 --- id3[id] </pre>
;	type \rightarrow integer	<pre> graph TD list1[list] --- list2[list] list1 --- comma1['] list1 --- id1[id] list2 --- var[var] list2 --- id2[id] list1 --- colon1['] list1 --- id3[id] type[type] --- integer1[integer] </pre>
empty	D \rightarrow var list : type ;	<pre> graph TD D[D] --- list1[list] D --- colon2['] D --- type1[type] D --- semicolon['] list1 --- var[var] list1 --- list2[list] list1 --- comma2['] list1 --- id1[id] list2 --- list3[list] list2 --- id2[id] list3 --- var2[var] list3 --- id3[id] type1 --- integer[integer] </pre>

The sentential forms that were produced during the parse are as given below. In each sentential form the rhs of the rule that is used for reduction has been highlighted.

```

var id , id : integer ;
var list , id : integer ;
var list : integer ;
var list : type ;
D

```

The sentential forms, as you can easily verify, happen to be a right most derivation in the reverse order.

Bottom-Up parsing and rightmost derivation

Working out a rightmost derivation in reverse turns out to be a natural choice for these parsers.

1. Right most sentential forms have the property that all symbols beyond the rightmost nonterminal are terminal symbols.
2. If for a string of terminals w , there exists a n -step rm-derivation from S , that is,
 $S = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n = w$; then given α_n it should be possible to construct α_{n-1} .
3. The $(n-1)^{\text{th}}$ right sentential form, denoted by α_{n-1} above, contains exactly one nonterminal, say A , but the position of A in this sentential form is not known. However, examining the tokens of w (or α_n) from left to right, one at a time, till the rhs of A , say β , is found, and then performing a reduction by $A \rightarrow \beta$ should give us α_{n-1} .
4. In general, the construction of a previous right most sentential form from a given right most sentential form is similar in spirit. The essence of bottom up parsing is to identify the rhs of a rule in the sentential forms.
5. It might be instructive to examine why it is not easy for a bottom-up parser to use a leftmost derivation in reverse.

Example : Consider the expression grammar whose rules are given by

$$\begin{aligned} E &\rightarrow E+T \mid T \\ T &\rightarrow T*F \mid F \\ F &\rightarrow \text{id} \mid (E) \end{aligned}$$

along with the input to be parsed : $\text{id} + \text{id} * \text{id}$

Leftmost derivation of this sentence is :

$$\begin{aligned} E &\Rightarrow_{\text{lm}} E + T \Rightarrow_{\text{lm}} T + T \Rightarrow_{\text{lm}} F + T \Rightarrow_{\text{lm}} \text{id} + T \Rightarrow_{\text{lm}} \text{id} + T * F \Rightarrow_{\text{lm}} \text{id} + F * F \\ &\Rightarrow_{\text{lm}} \text{id} + \text{id} * F \Rightarrow_{\text{lm}} \text{id} + \text{id} * \text{id} \end{aligned}$$

Observe the following from the sequence of derivation above.

- Consider the input sentence $\text{id} + \text{id} * \text{id}$. There are three instances of **id** in the given input, that matches with the rhs of the rule, $F \rightarrow \text{id}$
- Reducing the first **id** in $\text{id} + \text{id} * \text{id}$ produces $F + \text{id} * \text{id}$ which is not the previous leftmost sentential form, $\text{id} + \text{id} * F$
- Reducing the second **id** in, $\text{id} + \text{id} * \text{id}$, produces $\text{id} + F * \text{id}$ which is also not the previous leftmost sentential form, $\text{id} + \text{id} * F$
- In order to produce the previous sentential, only the third **id** in, $\text{id} + \text{id} * \text{id}$, should be used in the reduction which yield the previous leftmost sentential form, $\text{id} + \text{id} * F$
- In summary, the parser would have several candidates for reduction (only one correct) and it is not easy to identify the right one.

Recall that the basic steps of a bottom-up parser are

1. Reduce the input string to the start symbol by performing a series of reductions.
2. The sentential forms produced while parsing must trace out a rm-derivation in reverse.

There are two main considerations to carry out a reduction,

- i. to identify a substring within a rm-sentential form which matches the rhs of a rule (there may be several such)
- ii. when this substring is replaced by the lhs of the matching rule, it must produce the previous rm-sentential form. Such substrings, being central to bottom up parsing, have been given the name **handle**. Bottom up parsing is essentially the process of detecting handles and using them in reductions. Different bottom-up parsers use different methods for handle detection.

Definition of a Handle

- A **handle** of a **right sentential form**, say γ , is a production rule $A \rightarrow \beta$, and a **position of β** in γ such that when β is replaced by A in γ , the resulting string is the **previous right sentential form** in a rightmost derivation of γ from S , the start nonterminal.
- Formally, if $S \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha \beta w$, then rule $A \rightarrow \beta$, in the position following α in γ , is a handle of γ (that is $\alpha\beta w$).
- It should be clear that only terminal symbols can appear to the right of a handle in a rightmost sentential form.

Shift-Reduce Parser

A parsing method that directly uses the principle of bottom up parsing outlined above is known as a shift reduce parser. It is a generic name for a family of bottom- up parsers that employ this strategy.

A shift reduce parser requires the following data structures,

- a buffer for holding the input string to be parsed,
- a data structure for detecting handles (a stack happens to be adequate), and
- a data structure for storing and accessing the lhs and rhs of rules.

The parser operates by applying the steps 1 and 2 repeatedly, till a situation given by either of steps 3 or 4 is found.

Step 1. Moving symbols from the input buffer onto the stack, one symbol at a time. This move is called a **shift** action.

Step 2. Checking whether a handle occurs in the stack; if a handle is detected then a reduction by an appropriate rule is performed (which includes popping the rhs of a rule from the stack and then pushing the lhs of the rule), this move is called a **reduce** action.

Step 3. If the stack contains the start symbol only and input buffer is empty, then we announce a successful parse. This move is known as **accept** action.

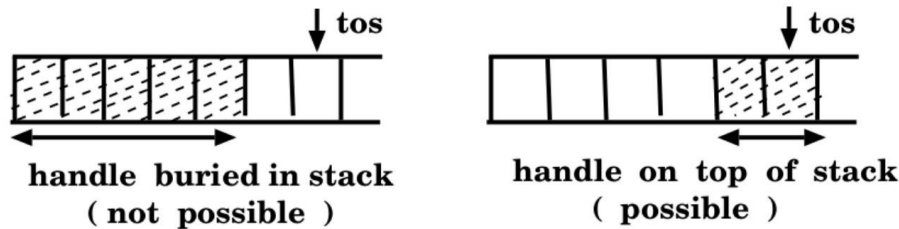
Step 4. A situation when the parser can neither shift nor reduce nor accept, it declares an error, which is known as **error** action and halts.

Remarks On Shift Reduce Parsing

1. Appending the stack contents with the unexpended input produces a rightmost sentential form.
2. Shift reduce parser as discussed above, is difficult to implement, since the details of when to shift and how to detect handles are not specified clearly.

3. A property that relates to the occurrence of handles in the parser stack is exploited by shift reduce parsers. If a handle can lie anywhere within the stack then handle detection is going to be expensive since all possible contiguous substrings in the stack are potential candidates for a handle.

4. It can be proved that a handle in a shift reduce parser always occurs on top of the stack (that is, the right end of the handle is the top of stack) and is never buried inside the stack.



Example : If the stack contents happens to be αAz , with z on tos , then possible handles are :

$z \quad Az \quad \alpha Az$

while substrings that are not handles are: $\alpha \quad \alpha A$

Handle Detection In Shift Reduce Parsing

Result : Handle in a shift reduce parser is never buried inside the stack.

Proof Outline : The result is easily verified for the last sentential form. Let $a_1 a_2 \dots a_n$ be the sentence to be parsed, and

$$S \Rightarrow_{\text{rm}}^* a_1 a_2 \dots a_i A a_{i+k+1} \dots a_n \Rightarrow_{\text{rm}} a_1 a_2 \dots a_n$$

The last two sentential forms (say m and $m-1$) are shown above, and $A \rightarrow a_{i+1} \dots a_{i+k}$ is the rule used. In this case shifting symbols upto a_{i+k} in the sentence, $a_1 a_2 \dots a_n$, would expose the handle on top of the stack . Similar arguments hold for the other possibilities of placement of A .

In the general case, consider the stack contents at some intermediate stage of parsing , say j^{th} sentential form, and assume the handle is on top of stack (tos) as shown below.

Stack	Input
αB	$x y z$

B is currently on tos (obtained after handle pruning from the $(j+1)^{\text{th}}$ sentential form). In order to construct the $(j-1)^{\text{th}}$ sentential form, the handle in the j^{th} sentential has to be detected. There are two possible forms for the $(j-1)^{\text{th}}$ sentential.

$(j-1)^{\text{th}}$ sentential	j^{th} sentential	Rule used
1. $\alpha B x A z$	$\alpha B x y z$	$A \rightarrow y$
2. $\delta A y z$	$\alpha B x y z$	$A \rightarrow \beta B x$

In case 1 above, after shifting x and y to the stack, the stack content will be $\alpha B x y$, and the handle y would be found on tos as claimed.

In case 2, after shifting x , the stack content will be $\alpha B x$, and if $\alpha = \delta \beta$, then stack contents are $\delta \beta B x$, and in this situation also the handle is on the top of stack, reiterating the fact that shifting of zero or more symbols allows the handle to occur on tos and get detected. This completes the proof outline.

The major implications of the above theoretical result are the following.

- Right end of a handle is guaranteed to occur at the top of the stack
- Left end of the handle would have to be found by examining as many symbols in the stack as present in the rhs of the rule.
- Search space for a handle is reduced considerably, at most linear in the size of the stack.
- Justifies the use of stack for detecting handles.

Illustrative Example : Consider the following extended grammar for expressions.

1, 2, 3) $E \rightarrow E + T \mid E - T \mid T$

4, 5, 6) $T \rightarrow T * F \mid T / F \mid F$

7, 8) $F \rightarrow P \uparrow F \mid P$

9, 10) $P \rightarrow -P \mid Q$

11, 12) $Q \rightarrow (E) \mid \text{id}$

The grammar rules are written by taking note of the operator properties. The symbol \uparrow denotes the exponentiation operator which is given to be right associative. There are two minus operators, the unary minus and the binary minus. Except for \uparrow , all the other operators are left associative. The operators, in the order from higher to lower precedence are : {unary minus ($-$); \uparrow ; $[\ast, /]$; $[\ast, /]$, where operators within square braces have the same precedence. The grammar given above is unambiguous and uses the empirical rules mentioned earlier – a) one nonterminal for every class of operators of same precedence, b) left or right recursive grammar for an operator that is respectively left or right associative.

Let us try an intuitive parsing of the string : $- a - b \ast c \uparrow d \uparrow e$ using a shift reduce parser. The parser shifts a token to the stack if the stack contents are part of a handle that has been partially seen. It reduces when a handle appears on tos. The moves of such a parser are illustrated as parsing proceeds. We show the stack contents with the tos marker at its right end. The input just before the parser move is shown. The parser action on the given configuration (stack, nexttoken) is mentioned. The execution of this action may either change the stack or the nexttoken or both. The new configuration after the action is shown in the next row in the following table.

The initial configuration comprises an empty stack (the symbol $\$$ is placed on the stack to indicate its bottom) and the input stream is appended with the symbol $\$$ to detect the end of input. The tos element is highlighted for ease in readability. The nexttoken is highlighted for the same reason.

Move	Stack contents	Input stream	Parser Action	Remarks
1	\$	- a - b * c \uparrow d \uparrow e \$	Shift -	Both input and stack change. The token - is part of rhs of rule 9

Move	Stack contents	Input stream	Parser Action	Remarks
2	\$ -	a - b * c ↑ d ↑ e \$	Shift a	Both input and stack change.
3	\$ - id	- b * c ↑ d ↑ e \$	Reduce by rule 12	Stack changes
4	\$ - Q	- b * c ↑ d ↑ e \$	Reduce by rule 10	Stack changes
5	\$ - P	- b * c ↑ d ↑ e \$	Reduce by rule 9	Stack changes
6	\$ P	- b * c ↑ d ↑ e \$	Reduce by rule 8	Stack changes
7	\$ F	- b * c ↑ d ↑ e \$	Reduce by rule 6	Stack changes
8	\$ T	- b * c ↑ d ↑ e \$	Reduce by rule 3	Stack changes
9	\$ E	- b * c ↑ d ↑ e \$	Shift -	Both change
10	\$ E -	b * c ↑ d ↑ e \$	Shift b	Both change
11	\$ E - id	* c ↑ d ↑ e \$	Reduce by rule 12	Stack changes
12	\$ E - Q	* c ↑ d ↑ e \$	Reduce by rule 10	Stack changes
13	\$ E - P	* c ↑ d ↑ e \$	Reduce by rule 8	Stack changes
14	\$ E - F	* c ↑ d ↑ e \$	Reduce by rule 6	Stack changes
15	\$ E - T	* c ↑ d ↑ e \$	Shift *	Both change
16	\$ E - T *	c ↑ d ↑ e \$	Shift c	Both change
17	\$ E - T * id	↑ d ↑ e \$	Reduce by rule 12	Stack changes
18	\$ E - T * Q	↑ d ↑ e \$	Reduce by rule 10	Stack changes
19	\$ E - T * P	↑ d ↑ e \$	Shift ↑	Both change
20	\$ E - T * P ↑	d ↑ e \$	Shift d	Both change
21	\$ E - T * P ↑ id	↑ e \$	Reduce by rule 12	Stack changes
22	\$ E - T * P ↑ Q	↑ e \$	Reduce by rule 10	Stack changes
23	\$ E - T * P ↑ P	↑ e \$	Shift ↑	Both change
24	\$ E - T * P ↑ P ↑	e \$	Shift e	Both change
25	\$ E - T * P ↑ P ↑ id	\$	Reduce by rule 12	Stack changes
26	\$ E - T * P ↑ P ↑ Q	\$	Reduce by rule 10	Stack changes
27	\$ E - T * P ↑ P ↑ P	\$	Reduce by rule 8	Stack changes
28	\$ E - T * P ↑ P ↑ F	\$	Reduce by rule 7	Stack changes
29	\$ E - T * P ↑ F	\$	Reduce by rule 7	Stack changes
30	\$ E - T * F	\$	Reduce by rule 4	Stack changes
31	\$ E - T	\$	Reduce by rule 2	Stack changes
32	\$ E	\$	Accept	Successful Parse

Explanation of a few selected moves of the shift reduce parser in the Example. Many of the parser moves may appear to be like black magic because there were alternative moves that were not taken. The fact is that the moves are not all adhoc, instead they are carefully guided based on analyses of the grammar rules. Insights with reasons about a few selected moves, chosen by the parser, from the above table are presented below.

1. Why did the parser in move 3 choose reduce by rule 12 instead of shifting the lookahead symbol ‘-’ to the stack ? The stack content at that point was “- id”. The only rule whose rhs matches partially with the stack content is “ $P \rightarrow -P$ ”. The parser therefore knows that to get to this handle, id has to be reduced to P, if such is possible. It therefore chooses the rule 12 to reduce “id” to “Q” resulting in the string “- Q” in the stack. However since “- Q” is not a handle, parser reduces this string “-id” by a series of reductions, using the rules, $Q \rightarrow id$ and $P \rightarrow Q$. As the handle “- P” gets exposed at tos in move 5 the parser uses the rule “ $P \rightarrow -P$ ” for reduction.
2. In move 6, the parser has the configuration (P, -) with P on tos and “-” as the lookahead. The parser does not shift “-”, since there is no handle with “P -” as a substring. However the parser is aware that “E - T” is a handle because of rule 2. Therefore it initiates a series of reductions, $F \rightarrow P$, $T \rightarrow F$, $E \rightarrow T$, via the moves 6 to 8 to ensure that the part “E”, of the handle “E -”, has been placed on the stack.

Reasoning for the other parser moves are left as an exercise for the reader.

Parse Tree Construction

A parse tree can be easily constructed by a shift reduce parser as it proceeds with parsing.

- Let the stack be augmented to include an extra field for each entry, a pointer to a tree associated with the symbol, on the stack. The code for tree construction can then be easily added to the basic moves of the parser as outlined below.
- shift a ; create a leaf labeled a; the root of this tree is associated with the stack symbol, a
- reduce by the rule : $A \rightarrow X_1 X_2 \dots X_n$; A new node labeled A is created ; sub-trees rooted at X_1, \dots, X_n are made its children in the left to right order. The pointer for this tree rooted at A is stored along with A in the stack.
- reduction by a rule, $A \rightarrow \epsilon$, is handled by creating a tree with root labeled A which has a leaf labeled with ϵ , as its only child. The pointer to the root of this tree is saved along with A in the stack.

Limitations Of Shift Reduce Parser

Q. What kind of grammars (languages) would not admit a shift reduce parser?

Consider the following situations.

1. a handle , say β , occurs at tos; the nexttoken is a, and βa happens to be another handle. The parser has two options for proceeding further.
 - Reduce the handle using $A \rightarrow \beta$. In such a case, the other handle , $B \rightarrow \beta a$, may not get exposed for reduction as β is not present in the stack anymore, and hence this longer handle may be missed for good.
 - Ignore the handle β present on the stack, shift a on to the stack and continue parsing. In this case, reduction using $B \rightarrow \beta a$ would be possible, because this handle may appear on the stack soon in the future. The price to pay is that of missing the handle β .

The situation above is described in the parser as a **shift reduce** conflict.

- The handle β , which has just appeared on the tos is such that it matches with the rhs of two or more rules, say $A \rightarrow \beta$ and $B \rightarrow \beta$. The the parser in this case has two or more reduce possibilities. This situation is known as a **reduce reduce** conflict.

Q. How do shift reduce parsers handle shift reduce and reduce reduce conflicts ?

The nexttoken is used by the parser to prefer one move over the other. The common wisdom is to choose shift (over reduce) in the case of a shift reduce conflict. The rationale given is that a longer handle is more desirable than the shorter one, since it is a substring of the longer handle. In the event of a reduce reduce conflict, parser designer specifies a tie breaking method to select one among the competing rules.

With this background information about a shift reduce parser, we begin the design of a family of deterministic shift reduce parsers, known as LR parsers.

LR Parsers : Some Facts

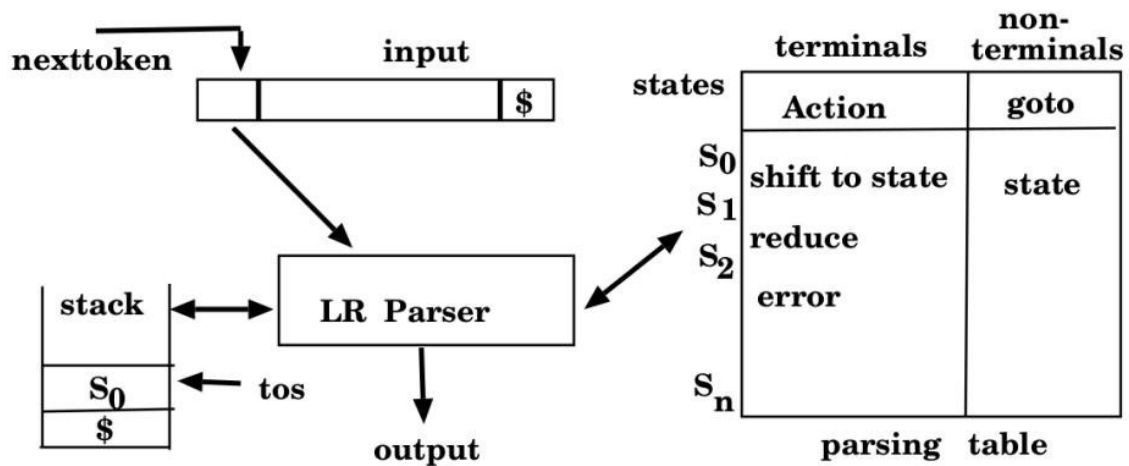
1. All the parsers in the family are shift reduce parsers, the basic parsing actions are shift, reduce, accept and error.
2. These are the most powerful of all deterministic parsers in practice; they subsume the class of grammars that can be parsed by predictive parsers.
3. LR parsers can be written to recognize most of the PL constructs which can be expressed through a cfg.
4. The overall structure of all the parsers is the same. All are table driven parsers, the information content in the respective tables distinguish the members of the family,

$$\text{SLR}(1) \leq \text{LALR}(1) \leq \text{LR}(1)$$

where $p \leq q$ means that q parses a larger class of grammars as compared to that parsed by p .

Structure of a LR parser

The components of a LR parser and their interaction is given in the figure. The parser, apart from grammar symbols, uses extra symbols (called states). While both grammar and state symbols can appear on the stack, the parsers are so designed that tos , is always a state.



Top of Stack (tos)	nexttoken	Parser Action	Realization of Action
State j (S_j)	a	Shift to state i (si)	push a; push S_i ; continue
	a	Reduce by rule j (rj)	$r_j : A \rightarrow \alpha$; $ \alpha = r$ (rhs has r symbols) pop 2r symbols from stack; tos = S_k (state on tos after popping) let goto [S_k, a] be S_m ; push A; push S_m ; continue
	\$	Accept (acc)	Successful parse; Halt
	a	error	Issue error message and handle error

Working of a LR Parser

Explanation Of Terms Used In Figure

The parser uses two data structures prominently.

1. A stack which contains strings of the form : $s_0X_1s_1 X_2 \dots X_ms_m$, where X_i is a grammar symbol and s_i is a special symbol called a state.
2. A parsing table which comprises two parts, usually named as, Action and Goto.
 - The Action part is a table of size $n \times m$ where n is the total number of states of the parser and $m = |T|$ (including \$). The possible entries are

- a) s_i which means shift to state i
- b) r_j which stands for reduce by the j^{th} rule,
- c) accept
- d) error

- The Goto part of the parsing table is of size $n \times p$, where $p = |N|$. The only interesting entries are state symbols.

The LR parser is a driver routine which

- i) initializes the stack with the start state and calls scanner to get a token (nexttoken).
- ii) For any configuration, as indicated by (tos , nexttoken), it consults the parsing table and performs the action specified there.
- iii) The goto part of the table is used only after a reduction.
- iv) The parsing continues till either an error or accept entry is encountered.

Simple LR(1) Parser

We start by examining the parser of the family which is known as Simple LR(1) or SLR(1) parser. SLR(1) Parsing Table for an expression grammar is given below. The working of this parser for a sample input is shown after the parsing table. Consider the simple expression Grammar again.

- 1, 2) $E \rightarrow E + T \mid T$
- 3, 4) $T \rightarrow T * F \mid F$
- 5, 6) $F \rightarrow (E) \mid \text{id}$

SLR(1) Parsing table for this grammar follows.

State	Action						Goto		
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Few key observations on the table structure and its contents for the given example. The table is of size 11 x 9, where the 11 rows record the details of each of the 11 states of the SLR(1) automaton, numbered from 0 to 10. The Action part of the table has 6 columns, one for each of 5 terminals of the grammar and an additional column for \$ (end-of-input-marker). The Goto part of the table has 3 columns, one for each of the nonterminals of G. To summarize, the size of SLR(1) table of size $n \times m$, where n is the number of states, and $m = |T| + |N| + 1$.

We shall explain later how the SLR(1) parsing table was constructed for this grammar. The use of table in parsing is illustrated with an example.

To illustrate the working of the parsing table on an input, consider the string : $a + b * (c + d) * e$

The 1st column counts the moves made by a SLR(1) parser. The 2nd column gives the stack contents with the tos at the right end, the 3rd column gives the unexpended input with the leftmost symbol as the current token, the 4th and 5th columns give the Action and Goto table entries respectively. The table entries are to be read in the following manner.

- Given the stack contents and the current token in the input, we consult the table with the configuration, [tos, nexttoken]. When tos is a state symbol, we consult the Action table.
- If the Action table entry is a shift move, the nexttoken is shifted to the stack and the new state is placed on tos.
- When the Action table entry is reduce by a rule, say $A \rightarrow \alpha$, twice the number of symbols present in the rhs α , that is $2*|\alpha|$ symbols are popped off the stack, $|\alpha|$ symbols of the rule and $|\alpha|$ state symbols that follow the grammar rules. After popping $2*|\alpha|$ symbols from the stack, a state symbol say t is exposed on the tos. The lhs nonterminal A is pushed on the stack after the state symbol t so that A now becomes the tos. A reduce results in a nonterminal getting placed on tos. This is the only situation when tos is not a state symbol and this needs to be rectified which is done by the next move.
- The Goto table is consulted for the entry $\text{Goto}[t, A]$ which gives a state say u . Then u is pushed on the stack with u becoming the tos, restoring a state on tos.

The details are presented in the following table.

Moves	Stack	Input	Action Table Entry	Goto Table Entry
1 (initial)	\$ 0	$a + b * (c + d) * e \$$	Shift to state 5	
2	\$ 0 id 5	$+ b * (c + d) * e \$$	Reduce by rule 6	
3	\$ 0 F	$+ b * (c + d) * e \$$		$\text{Goto}[0, F] = 3$
4	\$ 0 F 3	$+ b * (c + d) * e \$$	Reduce by rule 4	
5	\$ 0 T	$+ b * (c + d) * e \$$		$\text{Goto}[0, T] = 2$
6	\$ 0 T 2	$+ b * (c + d) * e \$$	Reduce by rule 2	
7	\$ 0 E	$+ b * (c + d) * e \$$		$\text{Goto}[0, E] = 1$
8	\$ 0 E 1	$+ b * (c + d) * e \$$	Shift to state 6	
9	\$ 0 E 1 + 6	$b * (c + d) * e \$$	Shift to state 5	
10	\$ 0 E 1 + 6 id 5	$* (c + d) * e \$$	Reduce by rule 6	
11	\$ 0 E 1 + 6 F	$* (c + d) * e \$$		$\text{Goto}[6, F] = 3$
12	\$ 0 E 1 + 6 F 3	$* (c + d) * e \$$	Reduce by rule 4	

Moves	Stack	Input	Action Table Entry	Goto Table Entry
13	\$ 0 E 1 + 6 T	* (c + d) * e \$		Goto[6, T] = 9
14	\$ 0 E 1 + 6 T 9	* (c + d) * e \$	Shift to state 7	
15	\$ 0 E 1 + 6 T 9 * 7	(c + d) * e \$	Shift to state 4	
16	\$ 0 E 1 + 6 T 9 * 7 (4	c + d) * e \$	Shift to state 5	
17	\$ 0 E 1 + 6 T 9 * 7 (4 id 5	+ d) * e \$	Reduce by rule 6	
18	\$ 0 E 1 + 6 T 9 * 7 (4 F	+ d) * e \$		Goto[4, F] = 3
19	\$ 0 E 1 + 6 T 9 * 7 (4 F 3	+ d) * e \$	Reduce by rule 4	
20	\$ 0 E 1 + 6 T 9 * 7 (4 T	+ d) * e \$		Goto[4, T] = 2
21	\$ 0 E 1 + 6 T 9 * 7 (4 T 2	+ d) * e \$	Reduce by rule 2	
22	\$ 0 E 1 + 6 T 9 * 7 (4 E	+ d) * e \$		Goto[4, E] = 8
23	\$ 0 E 1 + 6 T 9 * 7 (4 E 8	+ d) * e \$	Shift to state 6	
24	\$ 0 E 1 + 6 T 9 * 7 (4 E 8 + 6	d) * e \$	Shift to state 5	
25	\$ 0 E 1 + 6 T 9 * 7 (4 E 8 + 6 id 5) * e \$	Reduce by rule 6	
26	\$ 0 E 1 + 6 T 9 * 7 (4 E 8 + 6 F) * e \$		Goto[6, F] = 3
27	\$ 0 E 1 + 6 T 9 * 7 (4 E 8 + 6 F 3) * e \$	Reduce by rule 4	
28	\$ 0 E 1 + 6 T 9 * 7 (4 E 8 + 6 T) * e \$		Goto[6, T] = 9
29	\$ 0 E 1 + 6 T 9 * 7 (4 E 8 + 6 T 9) * e \$	Reduce by rule 1	
30	\$ 0 E 1 + 6 T 9 * 7 (4 E) * e \$		Goto[4, E] = 8
31	\$ 0 E 1 + 6 T 9 * 7 (4 E 8) * e \$	Shift to state 11	
32	\$ 0 E 1 + 6 T 9 * 7 (4 E 8) 11	* e \$	Reduce by rule 5	
33	\$ 0 E 1 + 6 T 9 * 7 F	* e \$		Goto[7, F] = 10
34	\$ 0 E 1 + 6 T 9 * 7 F 10	* e \$	Reduce by rule 3	
35	\$ 0 E 1 + 6 T	* e \$		Goto[6, T] = 9
36	\$ 0 E 1 + 6 T 9	* e	Shift to state 7	
37	\$ 0 E 1 + 6 T 9 * 7	e \$	Shift to state 5	
38	\$ 0 E 1 + 6 T 9 * 7 id 5	\$	Reduce by rule 6	
39	\$ 0 E 1 + 6 T 9 * 7 F	\$		Goto[7, F] = 10
40	\$ 0 E 1 + 6 T 9 * 7 F 10	\$	Reduce by rule 3	
41	\$ 0 E 1 + 6 T	\$		Goto[6, T] = 9
42	\$ 0 E 1 + 6 T 9	\$	Reduce by rule 1	
43	\$ 0 E	\$		Goto[0, E] = 1
44	\$ 0 E 1	\$	Accept	
Parsing terminates with success				

The parser recognizes the input as a valid string of the grammar after 44 moves.

Configuration of a LR Parser

A configuration of a LR parser is defined by a tuple, (stack contents , unexpended part of input).

Initial configuration is shown by $(s_0, a_1a_2 \dots a_n\$)$, where s_0 is the designated start state and the second component is the entire sentence to be parsed.

Let an intermediate configuration be given by $(s_0X_1s_1 \dots X_is_i, a_{j+1} \dots a_n\$)$, which shows that the state s_i is on top, and the unexpended input is, $a_{j+1} \dots a_n$, then resulting configuration

i) after a shift action is given by $(s_0X_1s_1 \dots X_is_ia_js_k, a_{j+1} \dots a_n\$)$ provided $Action[s_i, a_j] = s_k$; both stack and nexttoken change after the shift, and

ii) after a reduce action is given by $(s_0X_1s_1 \dots X_{i-r}s_{i-r}As, a_j \dots a_n\$)$, where $Action[s_i, a_j] = r_m$; and the rule $r_m : Action[s_i, a_j] : A \rightarrow \beta$, β has r grammar symbols and $goto(s_{i-r}, A) = s$. Only the stack changes here.

SLR(1) Parser Construction

We now study SLR(1) parsing table construction.

1. The relevant definitions are introduced first.
2. The construction process is then explained in full detail.
3. Theory of LR parsing which provides the basis for the construction process and also shows its correctness is addressed last.

Definition of LR(0) Item and Related Terms

- **LR(0) item** : An LR(0) item for a grammar G is a production rule of G with the symbol \bullet (read as dot or bullet) inserted at some position in the rhs of the rule.
- Example of LR(0) items : Consider the rule given below.

$decls \rightarrow decls\ decl$

the possible LR(0) items are :

I1 : $decls \rightarrow \bullet decls\ decl$

I2 : $decls \rightarrow decls \bullet decl$

I3 : $decls \rightarrow decls\ decl \bullet$

The rule $decls \rightarrow \epsilon$ has only one LR(0) item,

I4 : $decls \rightarrow \bullet$

- **Incomplete and Complete LR(0) items** : An LR(0) item is complete if the \bullet is the last symbol in the rhs, else it is an incomplete item.

For every rule, $A \rightarrow \alpha$, $\alpha \neq \epsilon$ there is only one complete item, $A \rightarrow \alpha\bullet$ but as many incomplete items as there are grammar symbols in the rhs.

Example : I3 and I4 are complete items and I1 and I2 are incomplete items.

- **Augmented Grammar** : From the grammar, $G = (N, T, P, S)$, we create a new grammar $G' = (N', T, P, S')$, where $N' = N \cup \{S'\}$, S' is the start symbol of G' and $P' = P \cup \{S' \rightarrow S\}$. Note that G' is equivalent to G , as we can formally prove that $L(G) = L(G')$.
- **Kernel and Nonkernel items** : An LR(0) item is called a kernel item, if the dot is not at the left end. However the item $S' \rightarrow \bullet S$, which is introduced by the new rule, $S' \rightarrow S$, is an exception and is defined to be a kernel item.

An LR(0) item which has dot at the left end is called a nonkernel item.

Example :

I1 : $\text{decls} \rightarrow \bullet \text{decls decl}$ is a nonkernel item

I2 : $\text{decls} \rightarrow \text{decls} \bullet \text{decl}$ is a kernel item

I3 : $\text{decls} \rightarrow \text{decls decl} \bullet$ is a kernel item

I4 : $\text{decls} \rightarrow \bullet$ is a nonkernel item

Note that if we augment the grammar above with a new start symbol, say D , and then add the rule $\{D \rightarrow \text{decls}\}$, then this rule produces the LR(0) item, $\{D \rightarrow \bullet \text{decls}\}$, which is a special kernel item, as stated above.

Canonical Collection of LR(0) Items

- Functions closure and goto : Two functions closure and goto are defined which are used to create a set of items from a given item.

Let U be the collection of all LR(0) items of a cfg G . The closure function, f , is of the form $f : U \rightarrow 2^U$ and is constructed as follows.

- i. $\text{closure}(I) = \{I\}$, for $I \in U$
- ii. If $A \rightarrow \alpha \bullet B\beta \in \text{closure}(I)$, then for every rule of the form $B \rightarrow \eta$, the item $B \rightarrow \bullet \eta$ is added (if it is not already there) to $\text{closure}(I)$.
- iii. Apply step (ii) above repeatedly till no more new items can be added to $\text{closure}(I)$.

It can be seen that $\text{closure}(I) \neq \{I\}$ only when I is an item in which a nonterminal immediately follows the dot.

- The goto function, g , has the form $g : U \times X \rightarrow 2^U$, where X is a grammar symbol. The function goto is defined using closure as follows.

$$\text{goto}(A \rightarrow \alpha \bullet X\beta, X) = \text{closure}(A \rightarrow \alpha X \bullet \beta).$$

Clearly $\text{goto}(I, X)$ for a complete item I is Φ , because there are no symbols to the right of the bullet (\bullet) for any X .

• Example of closure and goto sets construction

Consider the rule : $A \rightarrow Aa \mid b$ which gives 2 LR(0) items, $A \rightarrow \bullet Aa$ and $A \rightarrow \bullet b$

- Now $\text{closure}(A \rightarrow \bullet Aa) = \{ A \rightarrow \bullet Aa \}$, by rule (i). The item $A \rightarrow \bullet b$ is added to the closure by rule (ii) giving $\text{closure}(A \rightarrow \bullet Aa) = \{ A \rightarrow \bullet Aa, A \rightarrow \bullet b \}$, and since no other LR(0) can be added, after step (iii), the final content is $\text{closure}(A \rightarrow \bullet Aa) = \{ A \rightarrow \bullet Aa, A \rightarrow \bullet b \}$.
- Using the closure set constructed above, we get $\text{goto}(A \rightarrow \bullet Aa, A) = \text{closure}(A \rightarrow A \bullet a) = \{ A \rightarrow A \bullet a \}$

The functions $\text{closure}()$ and $\text{goto}()$ were constructed for a single LR(0) item. These sets can be constructed to a set S of LR(0) items by appropriate generalizations, as shown below.

$$\text{closure}(S) = \bigcup_{I \in S} \{ \text{closure}(I) \}; \text{ and } \text{goto}(S, X) = \bigcup_{I \in S} \{ \text{goto}(I, X) \}$$

Algorithm given below give pseudo code for computing both of these functions.

Given a cfg, the collection U of all LR(0) items is well defined.

A particular collection C of sets of items i. e., $C \in 2^U$, is of specific interest and is called the canonical collection of LR(0) items. This collection is constructed using $\text{closure}()$ and $\text{goto}()$ as given in the following Algorithm.

Canonical Collection of LR(0) Items

Algorithm for construction of sets of items

procedure items(G', C)

begin

$i = 0$; $I_0 = \{ \text{closure}(S' \rightarrow \bullet S) \}$; $C = I_i$;

repeat

for each set of items I_i in C and each grammar symbol X ,

such that $\text{goto}(I_i, X) \neq \Phi$ and $\text{goto}(I_i, X) \notin C$, **do**

$i++$; $I_i := \text{goto}(I_i, X)$;

$C := C \cup I_i$

until no more sets of items can be added to C

end


```

procedure closure(I); { I is a set of items }
begin
  repeat
    for each item  $A \rightarrow \alpha \bullet B\beta \in I$ 
      and each rule  $B \rightarrow \gamma$  such that  $B \rightarrow \bullet\gamma$  is not in I
    do add  $B \rightarrow \bullet\gamma$  to I;
  until no more items can be added to I ;
  return I ;
end

function goto(I, X)
begin
  goto items :=  $\Phi$  ;
  for each item  $A \rightarrow \alpha \bullet X\beta \in I$ ,
    such that  $A \rightarrow \alpha X \bullet \beta$  is not in goto items
  do add  $A \rightarrow \alpha X \bullet \beta$  to goto items
  goto := closure(goto items);
  return goto ;
end

```

Illustration of Construction of Canonical Collection

Consider the expression grammar augmented with $E' \rightarrow E$.

$$\begin{aligned}
 E' &\rightarrow E \\
 E &\rightarrow E + T \mid T \\
 T &\rightarrow T * F \mid F \\
 F &\rightarrow (E) \mid \text{id}
 \end{aligned}$$

I_0 is closure ($E' \rightarrow \bullet E$). The items $I_1 = \text{goto}(I_0, E)$ are added to I_0 . The last item in turn causes the addition of $T \rightarrow \bullet T * F$ and $T \rightarrow \bullet F$ to I_0 . The item $T \rightarrow \bullet F$ leads to the addition of $F \rightarrow \bullet (E)$ and $F \rightarrow \bullet \text{id}$. No more items can now be added and the collection I_0 is therefore complete and is shown in the figure that follows.

I_0 has several items having $\bullet\alpha$ combination for $\alpha = \{ E, T, F, (, \text{id} \}$. The set $\text{goto}(I_0, \alpha)$ is defined for all these values of α which in turn give rise to different collections as shown. For example, we choose the following names of the new collections arising out of I_0 . $I_1 = \text{goto}(I_0, E)$; $I_2 = \text{goto}(I_0, T)$; $I_3 = \text{goto}(I_0, F)$; $I_4 = \text{goto}(I_0, ()$; $I_5 = \text{goto}(I_0, \text{id})$, and so on. The rest of the collections are given below.

<div> $E' \rightarrow \bullet E$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet T * F$ $T \rightarrow \bullet F$ $F \rightarrow \bullet (E)$ $F \rightarrow \bullet \text{id}$ </div> <div>I_0</div>	<div> $E' \rightarrow E \bullet$ $E \rightarrow E \bullet + T$ </div> <div>$I_1 = \text{goto}(I_0, E)$</div>	<div> $E \rightarrow T \bullet$ $T \rightarrow T \bullet * F$ </div> <div>$I_2 = \text{goto}(I_0, T)$</div>	<div> $T \rightarrow F \bullet$ </div> <div>$I_3 = \text{goto}(I_0, F)$</div>	<div> $F \rightarrow (\bullet E)$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet T * F$ $T \rightarrow \bullet F$ $F \rightarrow \bullet (E)$ $F \rightarrow \bullet \text{id}$ </div> <div>$I_4 = \text{goto}(I_0, ()$</div>
	<div> $F \rightarrow \text{id} \bullet$ </div> <div>$I_5 = \text{goto}(I_0, \text{id})$</div>			

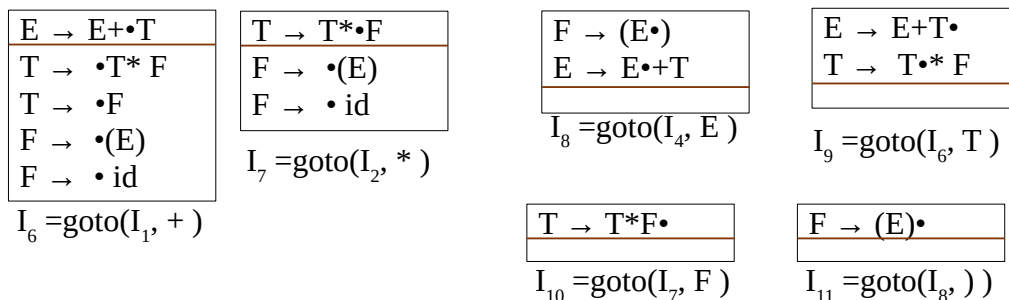


Figure : Canonical Collection of LR(0) items organized into states

Construction of SLR(1) Parsing Table

From the collection C constructed using the Algorithm, one can directly construct the SLR(1) parsing table. The procedure is described in the form of an algorithm.

Algorithm for SLR(1) Parsing Table

Input : Grammar G

Output : Action and goto part of the parsing table

1. From the input grammar G, construct an equivalent augmented grammar G'.
2. Use the algorithm for constructing FOLLOW sets to compute FOLLOW(A), $\forall A \in N'$.
3. Call procedure items(G', C) to get the desired canonical collection $C = \{I_0, I_1, \dots, I_n\}$.
4. Choose as many state symbols as the cardinality of C. We use numbers 0 through n to represent states.
5. The set I_i and its constituent items define the entries for state i of the Action table and the Goto table.

For each I_i , $0 \leq i \leq n$ do steps 5.1 through 5.6 given below.

5.1 If $A \rightarrow \alpha \bullet a \beta \in I_i$ and $\text{goto}(I_i, a) = I_j$, then $\text{Action}[i, a] = s_j$, i. e., shift to state j

5.2 If $A \rightarrow \alpha \bullet \in I_i$, then $\text{Action}[i, a] = r_j$, $\forall a \in \text{FOLLOW}(A)$ which means reduce by the j^{th} rule

$$A \rightarrow \alpha$$

5.3 If I_i happens to contain the item $S' \rightarrow S\bullet$, then $\text{Action}[i, \$] = \text{accept}$

5.4 All remaining entries of state i in the Action table are marked as error

5.5 For nonterminals A, such that $\text{goto}(I_i, A) = I_j$ create $\text{Goto}[i, A] = j$

5.6 The remaining entries in the Goto table are marked as error.

Note : The Goto entries of the state i of the parsing table are decided by the $\text{goto}(I_i, A)$ states, where A is a nonterminal.

The SLR(1) table for the canonical collection is partially constructed in the following.

Remarks : 1. initial or start state of the parser is the state corresponding to the set I_0 which contains the kernel item $S' \rightarrow \bullet S$

2. The parsing table is conflict free since there is no multiple entry (shift-reduce and/or reduce-reduce) in it.

Illustration of Table Construction Algorithm

The steps of the algorithm are traced out to fill up the first 3 states of the parsing table. Let us examine the LR(0) items in state 0.

$E' \rightarrow \bullet E$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet T * F$ $T \rightarrow \bullet F$ $F \rightarrow \bullet (E)$ $F \rightarrow \bullet id$	$E' \rightarrow E \bullet$ $E \rightarrow E \bullet + T$ $I_1 = \text{goto}(I_0, E)$	$E \rightarrow T \bullet$ $T \rightarrow T \bullet * F$ $I_2 = \text{goto}(I_0, T)$	$T \rightarrow F \bullet$ $I_3 = \text{goto}(I_0, F)$	$F \rightarrow (\bullet E)$ $E \rightarrow \bullet E + T$ $E \rightarrow \bullet T$ $T \rightarrow \bullet T * F$ $T \rightarrow \bullet F$ $F \rightarrow \bullet (E)$ $F \rightarrow \bullet id$
I_0	$I_5 = \text{goto}(I_0, id)$			$I_4 = \text{goto}(I_0, ()$

- There are 2 items with \bullet followed by a terminal, namely $F \rightarrow \bullet (E)$ and $F \rightarrow \bullet id$. These 2 items result in 2 shift moves in the Action part of table. The target states are dictated by $\text{goto}(I_0, id) = I_5$ and $\text{goto}(I_0, () = I_4$. The table at this point is drawn below.

State	Action						Goto		
	id	+	*	()	\$	E	T	F
0	s5			s4					

- There are 5 items where \bullet is followed by a nonterminal, namely $E' \rightarrow \bullet E$; $E \rightarrow \bullet E + T$; $E \rightarrow \bullet T$; $T \rightarrow \bullet T * F$; and $T \rightarrow \bullet F$. These items result in 3 entries in the Goto table determined by the states obtained using $\text{goto}(I_0, E) = I_1$; $\text{goto}(I_0, T) = I_2$ and $\text{goto}(I_0, F) = I_3$ which causes 3 entries in the Goto part of the table as shown below.

State	Action						Goto		
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3

- Since there are no complete LR(0) items (items with \bullet at the end) in I_0 , no reduce actions are possible in this state. The row for state 0 is complete at this point.

Let us consider filling up the entries of the table for state 1. This state has 2 LR(0) items, $E' \rightarrow E \bullet$ and $E \rightarrow E \bullet + T$. The item, $E \rightarrow E \bullet + T$, is a case of a terminal following the \bullet and hence causes an entry in the Action part determined by $\text{goto}(I_1, +) = I_6$, that is shift to state 6 on +.

State	Action						Goto		
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6							

The other LR(0) item, $E' \rightarrow E\bullet$, is a complete item and calls for reduce by this rule. This being a special reduce to the start symbol of the augmented grammar is the accept action which is always entered against \$. The table changes to the following at this point.

State	Action						Goto		
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			

This completes the processing for state 1. Note that there are no entries in Goto part of this table as there are no LR(0) items in state 1 where the \bullet is followed by a nonterminal.

As a final illustration, we consider state 2 and its contents to lay out the row for state 2. The LR(0) items in state 2 are $\{E \rightarrow T\bullet, T \rightarrow T\bullet *F\}$. The item, $T \rightarrow T\bullet *F$, causes a shift to the state goto($I_2, *$) which is I_7 . This entry for state 2 is placed into the table below.

State	Action						Goto		
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2			s7						

The other item, $E \rightarrow T\bullet$, is a complete item and calls for a reduce by the corresponding rule, which is rule 2. SLR(1) uses the FOLLOW() to enter the reduce actions. FOLLOW(E) is the set of 3 terminals $\{+,), \$\}$. Hence we place reduce by rule 2 in the Action part of the table for these 3 terminals. Since state 2 has no items with a nonterminal following the \bullet , the Goto part has no entries for this state. The table at this stage of construction has the following entries.

We can check that the table given earlier has exactly the same entries for the first 3 rows as shown below.

State	Action						Goto		
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			

SLR(1) Grammar and Parser

A grammar for which there is a conflict free SLR(1) parsing table is called a SLR(1) grammar and a parser which uses such a table is known as SLR(1) parser.

Q. How to detect shift-reduce conflicts and reduce-reduce conflicts in SLR(1) parser ?

- Step 5 of Algorithm 3.3 contains all the information. A shift- reduce conflict is detected when a state has

(a) a complete item of the form $A \rightarrow \alpha \bullet$ with $a \in \text{FOLLOW}(A)$, and also

(b) an incomplete item of the form $B \rightarrow \beta \bullet a \gamma$

- A reduce-reduce conflict is noticed when a state has two or more complete items of the form

(a) $A \rightarrow \alpha \bullet$

(b) $B \rightarrow \beta \bullet$, and

(c) $\text{FOLLOW}(A) \cap \text{FOLLOW}(B) \neq \Phi$

Having seen how to construct SLR(1) parsing table manually, we need to address the conceptual aspects of this method in order to answer questions of the following kind.

Exercise : Draw the SLR(1) automaton given in the Canonical Collection of LR(0) items given earlier in the figure, and also in the SLR(1) parsing table in the form of a directed graph. Then answer the following questions.

Q. What are the state symbols used by the parser and what information do they contain ?

Q. What is the significance of a path from start state to a given state ?

Q. How to produce a rightmost sentential while the parsing is in progress ?

End of Document

THEORY OF LR PARSING

The canonical collection of LR(0) items, discussed in detail in SLR(1) Parser, can also be represented by a directed labeled graph. Let the canonical collection of LR(0) items be represented as sets, $C = \{I_0, I_1, \dots, I_n\}$.

- A node of the graph, labeled I_i , is constructed for each member of C .
- For every nonempty collection given by $\text{goto}(I_i, X) = I_k$, for a grammar symbol X , a directed edge (I_i, I_k) is added to the graph labeled with X .
- The graph is a deterministic finite automaton with the node labeled I_0 as the start state and all other nodes treated as final states.
- The finite automaton associated with the collection of items for a sample declaration grammar is constructed brick by brick and shown in the following figures for illustration.

Construction of SLR(1) Automaton for Expression Grammar

Expression Grammar : 1) $E' \rightarrow E$ 2,3) $E \rightarrow E + T \mid T$ 4, 5) $T \rightarrow T * F \mid F$
6, 7) $F \rightarrow (E) \mid \text{id}$

1. E' is the new start symbol in the augmented grammar.
2. We begin with the first collection of LR(0) items, named as I_0 . The kernel item of I_0 is always, $E' \rightarrow \bullet E$ which relates the new start symbol with the old start symbol. To complete the collection of items in I_0 , find the $\text{closure}(E' \rightarrow \bullet E)$.
3. $\text{Closure}(E' \rightarrow \bullet E)$: since $\bullet E$ appears in the rhs of the LR(0) item, it results in including all rules of E with a \bullet at the first position in their rhs, namely, $E \rightarrow \bullet E + T$ and $E \rightarrow \bullet T$
4. The first LR(0) item is already closed and does not add a new item. However $\text{closure}(E \rightarrow \bullet T)$ due to the 2nd item, results in the addition of 2 more items, $T \rightarrow \bullet T * F \mid \bullet F$
5. Continuing further, $\text{closure}(T \rightarrow \bullet F)$ adds two more items, $F \rightarrow \bullet (E) \mid \bullet \text{id}$ The last 2 items do not have \bullet followed by a nonterminal and hence do not need to be closed.
6. The calculations performed above are summarized as follows :
 $I_0 = \text{closure}(E' \rightarrow \bullet E) = \{ E' \rightarrow \bullet E; E \rightarrow \bullet E + T \mid \bullet T; T \rightarrow \bullet T * F \mid \bullet F; F \rightarrow \bullet (E) \mid \bullet \text{id} \}$
 $\text{closure}(\bullet F) = \{ F \rightarrow \bullet (E) \mid \bullet \text{id} \};$ $\text{closure}(\bullet T) = \{ T \rightarrow \bullet T * F \mid \bullet F; F \rightarrow \bullet (E) \mid \bullet \text{id} \};$ and
 $\text{closure}(\bullet E) = \{ E \rightarrow \bullet E + T \mid \bullet T; T \rightarrow \bullet T * F \mid \bullet F; F \rightarrow \bullet (E) \mid \bullet \text{id} \}$
7. The last three closures were obtained as a byproduct. Since these closures are independent of the context, they can readily be reused in the calculations that follow. This collection is represented as a node, where the kernel item is separated from the items added by closure, by drawing a **red line** separating them.

Once a set of items is constructed, we need to determine other collections that are generated by it. For each item in the collection, in which \bullet is not at the right end, a new collection of set of items are constructed by moving the \bullet past the grammar symbol that follows it. Items in which \bullet is at the right end do not permit the \bullet to move any further.

For I_0 , we observe the following pairs of \bullet followed by a grammar symbol.

1. Two items with $\bullet E$, as in $\{E' \rightarrow \bullet E; E \rightarrow \bullet E + T\}$. The new collection when \bullet goes past E is created by placing the items $\{E' \rightarrow E\bullet; E \rightarrow E\bullet + T\}$ in the kernel of the new collection, named say as I_1 . This collection is also denoted by the notation, $\text{goto}(I_0, E)$. The collection of all items in I_1 is constructed by taking the closure of the kernel items of I_1 .
2. Two items with $\bullet T$, as in $\{E \rightarrow \bullet T; T \rightarrow \bullet T * F\}$. The new collection, when \bullet goes past T , is created by placing the items $\{E \rightarrow T\bullet; T \rightarrow T\bullet * F\}$ in the kernel of the new collection, named say as I_2 . This collection is denoted by the notation, $\text{goto}(I_0, T)$. The collection of all items in I_2 is constructed by taking the closure of the kernel items of I_2 .
3. One item with $\bullet F$, as in $\{T \rightarrow \bullet F\}$. The new collection, when \bullet goes past F , is created by placing the items $\{T \rightarrow F\bullet\}$ in the kernel of the new collection, named say as I_3 . This collection is denoted by the notation, $\text{goto}(I_0, F)$. The collection of all items in I_3 is constructed by taking the closure of the kernel items of I_3 ; however since the \bullet is at the right end in I_3 , closure is not applicable.
4. One item with $\bullet \text{id}$, as in $\{F \rightarrow \bullet \text{id}\}$. The new collection, when \bullet goes past id , is created by placing the items $\{F \rightarrow \text{id}\bullet\}$ in the kernel of the new collection, named say as I_4 . This collection is denoted by the notation, $\text{goto}(I_0, \text{id})$. The collection of all items in I_4 is constructed by taking the closure of the kernel items of I_4 ; however since the \bullet is at the right end in I_4 , closure is not applicable.
5. One item with $\bullet ($, as in $\{F \rightarrow \bullet (E)\}$. The new collection, when \bullet goes past $($, is created by placing the items $\{F \rightarrow (\bullet E)\}$ in the kernel of the new collection, named say as I_5 . This collection is denoted by the notation, $\text{goto}(I_0, ($). The collection of all items in I_5 is constructed by taking the closure of the kernel items of I_5 ; however since the \bullet is at the right end in I_5 , closure is not applicable.
6. All the 7 LR(0) items in I_0 have now been processed for movement of \bullet past the grammar symbols and the kernels of the 5 new collections constructed. At this stage of the construction, the collection I_0 and its $\text{goto}(I_0, X)$ collections are complete and a summary is captured in the following figure.

<table><tr><td>$E' \rightarrow \bullet E$</td></tr><tr><td>$E \rightarrow \bullet E + T \mid \bullet T$</td></tr><tr><td>$T \rightarrow \bullet T * F \mid \bullet F$</td></tr><tr><td>$F \rightarrow \bullet (E) \mid \bullet \text{id}$</td></tr></table> <p>Collection I_0</p>	$E' \rightarrow \bullet E$	$E \rightarrow \bullet E + T \mid \bullet T$	$T \rightarrow \bullet T * F \mid \bullet F$	$F \rightarrow \bullet (E) \mid \bullet \text{id}$	<p>Kernel of $I_1 = \text{Kernel}(\text{goto}(I_0, E)) = \{ E' \rightarrow E\bullet; E \rightarrow E\bullet + T \}$</p> <p>Kernel of $I_2 = \text{Kernel}(\text{goto}(I_0, T)) = \{ E \rightarrow T\bullet; T \rightarrow T\bullet * F \}$</p> <p>Kernel of $I_3 = \text{Kernel}(\text{goto}(I_0, F)) = \{ T \rightarrow F\bullet \}$</p> <p>Kernel of $I_4 = \text{Kernel}(\text{goto}(I_0, \text{id})) = \{ F \rightarrow \text{id}\bullet \}$</p> <p>Kernel of $I_5 = \text{Kernel}(\text{goto}(I_0, ()) = \{ F \rightarrow (\bullet E) \}$</p>
$E' \rightarrow \bullet E$					
$E \rightarrow \bullet E + T \mid \bullet T$					
$T \rightarrow \bullet T * F \mid \bullet F$					
$F \rightarrow \bullet (E) \mid \bullet \text{id}$					

The collection of LR(0) items, I_1 through I_5 are constructed and displayed in the following table. It can be seen that in I_1 , I_2 , I_3 and I_4 do not get any items added by closure as \bullet is not followed by a nonterminal. Collection I_5 has 6 more LR(0) items added because of $\text{closure}(E)$, which has already been precomputed.

$I_1 = \text{goto}(I_0, E)$	$I_2 = \text{goto}(I_0, T)$	$I_3 = \text{goto}(I_0, F)$	$I_4 = \text{goto}(I_0, \text{id})$	$I_5 = \text{goto}(I_0, ()$
<div> $E' \rightarrow E\bullet$ $E \rightarrow E\bullet + T$ <hr/> Φ </div>	<div> $E \rightarrow T\bullet$ $T \rightarrow T\bullet * F$ <hr/> Φ </div>	<div> $T \rightarrow F\bullet$ <hr/> Φ </div>	<div> $F \rightarrow \text{id}\bullet$ <hr/> Φ </div>	<div> $F \rightarrow (\bullet E)$ <hr/> $E \rightarrow \bullet E + T \mid \bullet T$ $T \rightarrow \bullet T * F \mid \bullet F$ $F \rightarrow \bullet (E) \mid \bullet \text{id}$ </div>

The SLR(1) automaton at this stage of construction is drawn below. The LR(0) items contained in the nodes are omitted for brevity. The SLR(1) parsing table is shown in the other part of this table. The entries are self-explanatory, with sk, denoting the parser action **shift to state k**. There is no reduce action in state 0, since no handles are seen in this state (no item with \bullet at the right end).

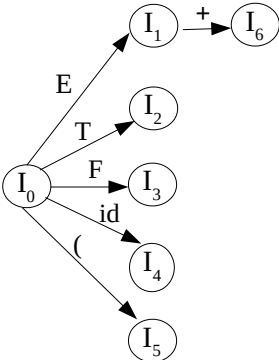
SLR(1) Automaton	SLR (1) Parsing Table										
	State	id	+	*	()	\$		E	T	F
	0	s4			s5				1	2	3

7. Let us explore the collection denoted by I_1 . The kernel items are $\{ E' \rightarrow E\bullet; E \rightarrow E\bullet + T \}$. The first item is a complete item and indicates a reduce by the associated rule 1, we usually denote this move as, “r1”, with the intended meaning as reduce by rule 1. The other item, $E \rightarrow E\bullet + T$, results in \bullet moving past E and forms the kernel item of the collection $\text{goto}(I_1, +)$, which is named as I_6 . The collection of items in I_6 , after its closure is $\{ E \rightarrow E + \bullet T; T \rightarrow \bullet T * F \mid \bullet F; F \rightarrow \bullet (E) \mid \bullet \text{id} \}$ and shown below.

$E' \rightarrow E\bullet$ $E \rightarrow E\bullet + T$ <hr/> Φ
I_1

$E \rightarrow E + \bullet T$ <hr/> $T \rightarrow \bullet T * F \mid \bullet F$ $F \rightarrow \bullet (E) \mid \bullet \text{id}$
I_6

We now have all the information to populate the row for state 1. Reduction by the rule, $E' \rightarrow E\bullet$, indicates that the entire input has been reduced to E' which is the unique accept move of this parser. This move is placed in the action[1, FOLLOW(E')] table for all terminals in FOLLOW(E'), which happens to be $\{\$ \}$ in this case.

SLR(1) Automaton	SLR (1) Parsing Table									
	State	id	+	*	()	\$	E	T	F
	0	s4			s5			1	2	3
	1		s6				acc			

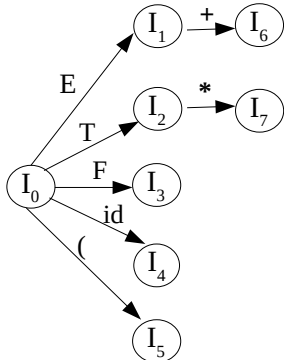
8. Let us construct the row for state 2 of the parsing table. This requires examining the LR(0) items in I_2 and also all the goto(I_2 , X) states. The items in I_2 are $\{E \rightarrow T\bullet; T \rightarrow T\bullet * F\}$. The first item calls for a reduce by rule 3 on all symbols in FOLLOW(E) = $\{+,), \$\}$. The 2nd item leads to a new state, goto(I_2 , *), named as I_7 . The LR(0) items in both the states are given below.

$E \rightarrow T\bullet$
$T \rightarrow T\bullet * F$
Φ

I_2

$T \rightarrow T\bullet * F$
$F \rightarrow \bullet(E) \mid \bullet id$

I_7

SLR(1) Automaton	SLR (1) Parsing Table									
	State	id	+	*	()	\$	E	T	F
	0	s4			s5			1	2	3
	1		s6				acc			
	2		r3	s7		r3	r3			

9. We now go to populate the table for the states 3, 4 and 5 together. The set of all LR(0) items in all the 3 states are reproduced below. The only item in I_3 , $T \rightarrow F \bullet$, calls for a reduce by rule 5 on all symbols in $\text{FOLLOW}(T) = \{*, +,), \$\}$. The only item in I_4 , $F \rightarrow \text{id} \bullet$, calls for a reduction by rule 7 on all symbols in $\text{FOLLOW}(F) = \text{FOLLOW}(T) = \{*, +,), \$\}$.

10. State I_5 has no complete items, no reduce action in state 5. Because of \bullet being followed by the 5 grammar symbols $\{E, T, F, (, \text{id}\}$, we construct the sets of LR(0) items, denoted by $\text{goto}(I_5, E)$, $\text{goto}(I_5, T)$, $\text{goto}(I_5, F)$, $\text{goto}(I_5, '(')$ and $\text{goto}(I_5, \text{id})$. The kernel of $\text{goto}(I_5, E)$ is $\{F \rightarrow (E \bullet), E \rightarrow E \bullet + T\}$ which is distinct from the kernels of all states constructed so far (though it has one item common with I_1), and hence we create a new state I_8 . It can be easily verified that kernel of $\text{goto}(I_5, T)$ is $\{E \rightarrow T \bullet\}$ which is same as $\text{kernel}(I_2)$, kernel of $\text{goto}(I_5, F)$ is $\{T \rightarrow F \bullet\}$ which is $\text{kernel}(I_3)$, kernel of $\text{goto}(I_5, '(')$ is $\{F \rightarrow (\bullet E)\}$ which is $\text{kernel}(I_5)$ and $\text{goto}(I_5, \text{id})$ is $\{F \rightarrow \text{id} \bullet\}$ which is $\text{kernel}(I_4)$.

$E \rightarrow T \bullet$	$T \rightarrow F \bullet$	$F \rightarrow \text{id} \bullet$	$F \rightarrow (\bullet E)$	$F \rightarrow (E \bullet)$
Φ	Φ	Φ	$E \rightarrow \bullet E + T \mid \bullet T$ $T \rightarrow \bullet T * F \mid \bullet F$ $F \rightarrow \bullet (E) \mid \bullet \text{id}$	$E \rightarrow E \bullet + T$
$I_2 = \text{goto}(I_0, T)$ $= \text{goto}(I_5, T)$	$I_3 = \text{goto}(I_0, F)$ $= \text{goto}(I_5, F)$	$I_4 = \text{goto}(I_0, \text{id})$ $= \text{goto}(I_5, \text{id})$	$I_5 = \text{goto}(I_0, '(')$ $= \text{goto}(I_5, '(')$	$I_8 = \text{goto}(I_5, E)$

SLR(1) Automaton	SLR (1) Parsing Table									
	State	id	+	*	()	\$	E	T	F
	0	s4			s5			1	2	3
	1		s6				acc			
	2		r3	s7		r3	r3			
	3		r5	r5		r5	r5			
	4		r7	r7		r7	r7			
	5	s4			s5			8	2	3

11. For I_6 , there are transitions on the grammar symbols $\{T, F, \text{id}, '('\}$. The kernel of $\text{goto}(I_6, T)$ has the items $\{E \rightarrow E + T \bullet; T \rightarrow T * F \bullet\}$ which is distinct from all the existing ones and is named I_9 , the kernels of the other $\text{goto}()$ are existing states, such as $\text{goto}(I_6, F)$ is I_3 , $\text{goto}(I_6, '(')$ is I_5 and $\text{goto}(I_6, \text{id})$ is I_4 .

12. For I_7 , there are transitions on 3 symbols $\{F, \text{id}, '('\}$. The kernel of $\text{goto}(I_7, F)$ is $\{T \rightarrow T * F \bullet\}$ which is new state and numbered as 10. The kernels of $\text{goto}(I_7, \text{id})$ and $\text{goto}(I_7, '(')$ are existing states 4 and 5 respectively.

13. For I_8 , there are transitions on the symbol $\{'\}$. The kernel of $\text{goto}(I_8, ')')$ is $\{F \rightarrow (E) \bullet\}$ which is new state and numbered as 11.

$E \rightarrow E + \bullet T$
$T \rightarrow \bullet T * F \mid \bullet F$
$F \rightarrow \bullet (E) \mid \bullet id$
I_6

$T \rightarrow T * \bullet F$
$F \rightarrow \bullet (E) \mid \bullet id$
I_7

$F \rightarrow (E \bullet)$
$E \rightarrow E + \bullet T$
Φ
I_8

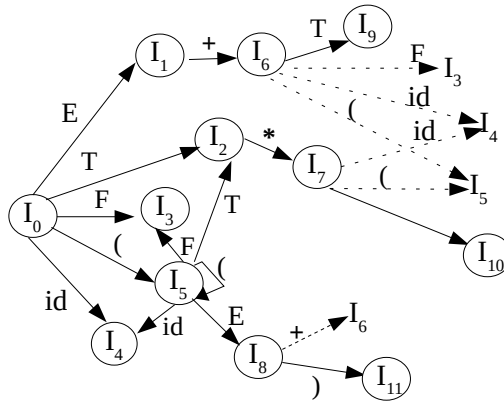
$E \rightarrow E + T \bullet$
$T \rightarrow T * \bullet F$
Φ
I_9

$T \rightarrow T * F \bullet$
Φ
I_{10}

$F \rightarrow (E) \bullet$
Φ
I_{11}

These computations are then inserted into the table filling up all entries of the states from 0 to 8.

SLR(1) Automaton	SLR (1) Parsing Table									
	State	id	+	*	()	\$	E	T	F
The automaton is drawn separately. Dotted arrows with labels are used to denote a transition to an existing state in order to avoid clutter in the graph. Solid edges with labels are used to denote a transition to a new state. The states are numbered in the order of creation during the construction process.	0	s4			s5			1	2	3
	1		s6				acc			
	2		r3	s7		r3	r3			
	3		r5	r5		r5	r5			
	4		r7	r7		r7	r7			
	5	s4			s5			8	2	3
	6	s4			s5				9	3
	7	s4			s5					10
	8		s6		s11					



14. Three more states, 9, 10 and 11 have been created so far, whose entries in the table have not been entered. The states 10 and 11, have a complete item each and hence have no transitions out of these nodes. State 10 calls for reduce by rule 4 for all symbols in $\text{FOLLOW}(T) = \{+, *,), \$\}$. Similarly state 11 calls for reduce by rule 6 for all symbols in $\text{FOLLOW}(F) = \{*, +,), \$\}$.

$E \rightarrow E + T \bullet$
$T \rightarrow T * \bullet F$
Φ
I_9

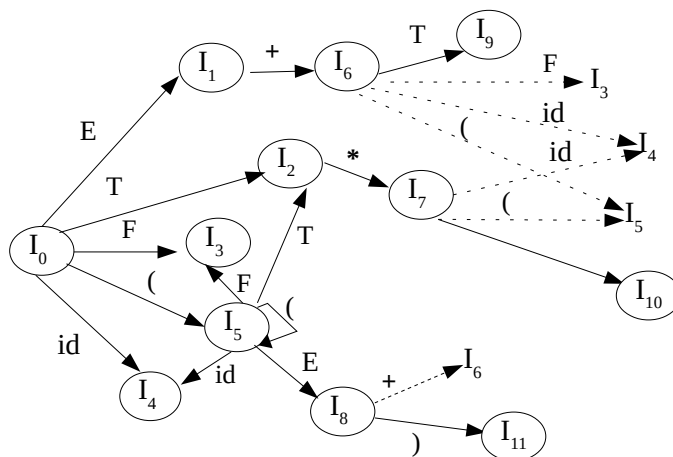
$T \rightarrow T * F \bullet$
Φ
I_{10}

$F \rightarrow (E) \bullet$
Φ
I_{11}

$T \rightarrow T * \bullet F$
$F \rightarrow \bullet (E) \mid \bullet id$
$\text{goto}(I_9, *) = I_7$

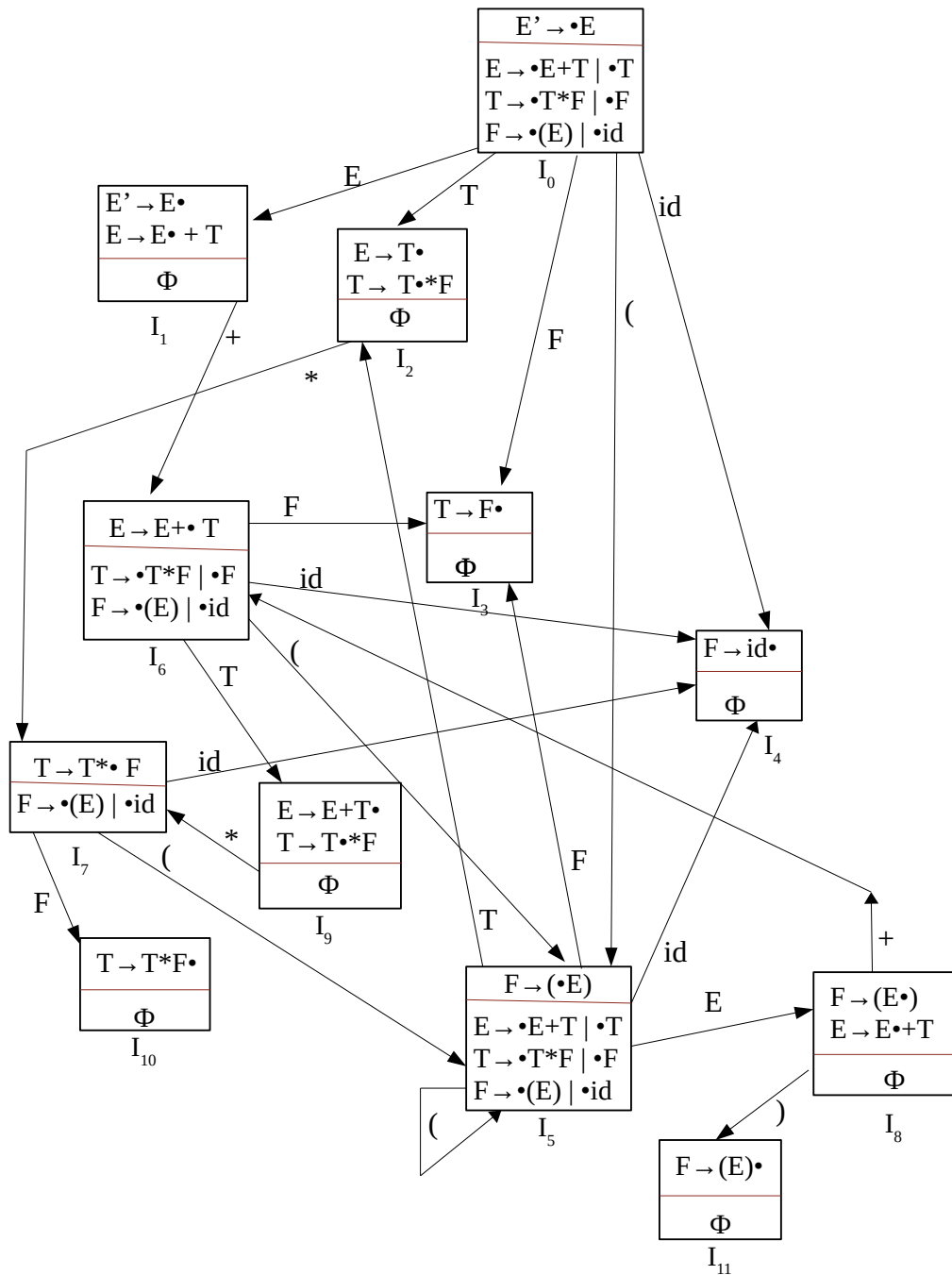
15. State 9 has a complete kernel item, $E \rightarrow E + T \cdot$, which calls for reduce by rule 2 for all symbols in $\text{FOLLOW}(E) = \{+,), \$\}$. The other kernel item is, $T \rightarrow T \cdot * F$, which results in the item, $T \rightarrow T \cdot * F$, in the kernel of $\text{goto}(I_9, *)$. However this state has already been constructed and is I_7 . Since no new state has been constructed and all the states have been processed for the LR(0) items contained therein, the construction of the SLR(1) automaton and its parsing table is complete. The final automaton and the parsing table is given below.
16. In the Action[] table, all blank entries denote a syntax error. In the Goto[] table, a blank entry is Not Applicable, as the parser can never reach such configuration. Why is the previous statement true?

SLR(1) Automaton	SLR (1) Parsing Table									
	State	id	+	*	()	\$	E	T	F
The automaton is drawn separately. Dotted arrows with labels are used to denote a transition to an existing state in order to avoid clutter in the graph. Solid edges with labels are used to denote a transition to a new state. The states are numbered in the order of creation during the construction process.	0	s4			s5			1	2	3
	1		s6				acc			
	2		r3	s7		r3	r3			
	3		r5	r5		r5	r5			
	4		r7	r7		r7	r7			
	5	s4			s5			8	2	3
	6	s4			s5				9	3
	7	s4			s5					10
	8		s6			s11				
	9		r2	s7		r2	r2			
	10		r4	r4		r4	r4			
	11		r6	r6		r6	r6			



A few important observations follow.

- The closure($\bullet A$), for every nonterminal A can be computed once and used repeatedly.
- The kernel items of a state s, together with the closure($\bullet A$) sets are sufficient to generate the kernel items of the goto(s, X).
- The SLR(1) automaton and the parsing table can be constructed efficiently by the two steps mentioned above.



Viable Prefix and Valid LR(0) Items

A viable prefix of a grammar is defined to be the prefixes of right sentential forms that do not contain any symbols to the right of a handle.

Only the kernel items of each state is shown in the above.

1. By adding terminal symbols to viable prefixes, rightmost sentential forms can be constructed.
2. Viable prefixes are precisely the set of symbols that can ever appear on the stack of a shift reduce parser.
3. A viable prefix either contains a handle or contains parts of one or more handles.
4. Given a viable prefix, if one could identify the set of potential handles associated with it then a recognizer for viable prefixes would also recognize handles.

The significance of LR(0) item can now be understood. A complete item corresponds to a handle while an incomplete item indicates the part of a handle seen so far, the symbols to the left of dot (\bullet), while the symbols to the right of dot are yet to be seen in the input processed thus far.

A LR(0) item $A \rightarrow \beta_1 \bullet \beta_2$ is defined to be valid for a viable prefix, $\alpha\beta_1$, provided

$$S \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha\beta_1\beta_2w \quad \dots (1)$$

That is from the start symbol S , we are able to reach a right sentential form, $\alpha\beta_1\beta_2w$, using the LR(0) item, $A \rightarrow \beta_1\bullet\beta_2$. The proof of validity of this item for the prefix $\alpha\beta_1$ is provided by the rm -derivation given in (1), which gives evidence about the fact that $\alpha\beta_1$ is a prefix of the right sentential form $\alpha\beta_1\beta_2w$ and of the rule, $A \rightarrow \beta_1\beta_2$, the part β_1 (before \bullet) has been seen, while the part β_2 is yet to appear.

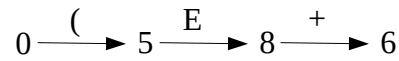
1. It is interesting to note that in above, if $\beta_2 = B\gamma$ where B is a nonterminal and $B \rightarrow \delta$ is a rule, then $B \rightarrow \bullet\delta$ is also a valid item for this viable prefix. This is because, when we continue with (1), we get $S \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha\beta_1\beta_2w = \alpha\beta_1B\gamma w \Rightarrow_{rm} \alpha\beta_1\delta\gamma w \quad \dots (2)$

The last right sentential form in (2), $\alpha\beta_1\delta\gamma w$, shows that for the viable prefix $\alpha\beta_1$, since δ appears immediately to its right, the item $B \rightarrow \bullet\delta$ is a valid item. However $B \rightarrow \delta\bullet$ is not a valid item for the viable prefix $\alpha\beta_1$. It may however be a valid item for a different viable prefix.

2. There could be several distinct items which are valid for same viable prefix γ , as was observed above. Therefore a mapping from Sets-of-Items \rightarrow viable-prefix is many to one.
3. Conversely, given a particular LR(0) item, how many distinct viable prefixes exist for which this item is valid. It so turns that this mapping is also many to one.
4. The parser would halt when the viable prefix S is seen, which formally stated is a move from $\bullet S$ to $S\bullet$. This is essentially the reason for augmenting the input grammar.

Illustrative Example : For the SLR(1) automaton given in the Figure above, let us work out these concepts and their relationships in some detail.

- Consider the path from state 0 to state 6 shown below. The labels on the edges traversed is “(E+”.



The items that are placed in state 6 are $\{E \rightarrow E+\bullet T; T \rightarrow \bullet T * F; T \rightarrow \bullet F; F \rightarrow \bullet (E); F \rightarrow \bullet id\}$.

Q. Why exactly these 5 LR(0) items are placed in state 6 and no other LR(0) items?

The answer lies in the understanding the relationship between viable prefix and valid items.

1. For the given path, the viable prefix is “(E +”. We wish to examine the rightmost derivations that include this viable prefix.
2. $E' \Rightarrow_{rm} E \Rightarrow_{rm} T \Rightarrow_{rm} F \Rightarrow_{rm} (E) \Rightarrow_{rm} (E + T)$... (1)
- $E' \Rightarrow_{rm} E \Rightarrow_{rm} T \Rightarrow_{rm} F \Rightarrow_{rm} (E) \Rightarrow_{rm} (E + T) \Rightarrow_{rm} (E + T * F)$... (2)
- $E' \Rightarrow_{rm} E \Rightarrow_{rm} T \Rightarrow_{rm} F \Rightarrow_{rm} (E) \Rightarrow_{rm} (E + T) \Rightarrow_{rm} (E + F)$... (3)
- $E' \Rightarrow_{rm} E \Rightarrow_{rm} T \Rightarrow_{rm} F \Rightarrow_{rm} (E) \Rightarrow_{rm} (E + T) \Rightarrow_{rm} (E + F) \Rightarrow_{rm} (E + (E))$... (4)
- $E' \Rightarrow_{rm} E \Rightarrow_{rm} T \Rightarrow_{rm} F \Rightarrow_{rm} (E) \Rightarrow_{rm} (E + T) \Rightarrow_{rm} (E + F) \Rightarrow_{rm} (E + id)$... (5)

Can you write another rightmost sentential form that includes the string “(E+” but is not included above ?

3. Consider derivation (1) above. $E' \Rightarrow_{rm}^* (E + T)$ has the string “(E+” as a prefix. The remaining string is “T)”. The LR(0) item, $E \rightarrow E+T$, has the part “E+” of the prefix already seen, hence the item, $E \rightarrow E+\bullet T$, is a valid item for “(E+”.
4. Consider derivation (2) above. $E' \Rightarrow_{rm}^* (E + T * F)$ has the string “(E+” as a prefix. The remaining string is “T * F)”. The LR(0) item, $T \rightarrow T * F$, has the no part of “E+” of the prefix, hence the item, $T \rightarrow \bullet T * F$, is also a valid item for “(E+”.
5. As a self assessment exercise, find the valid items for derivations (3) to (5).

Q. Why a particular LR(0) item, say $T \rightarrow \bullet F$, appears in multiple states, which is 0, 5, and 6 in the given automaton ?

The viable prefix of state 0 is ϵ (the null string). The rightmost derivation, $E' \Rightarrow_{rm} E \Rightarrow_{rm} T \Rightarrow_{rm} F$ is interpreted as, $\epsilon \mid F$, where the viable prefix is to the left of \mid . Since F is yet to be seen, $T \rightarrow \bullet F$ is a valid item for this state.

The viable prefix of state 6 is “E+”. The rightmost derivation, $E' \Rightarrow_{rm} E + T \Rightarrow_{rm} E + F$, can be interpreted as, $E+ \mid F$, where the viable prefix “E+” has been seen and F is expected to appear, showing that $T \rightarrow \bullet F$ is a valid item for this state.

The viable prefix of state 5 is $\{ '(', '(', '(((, '((((, \dots \}$, that is a sequence of one or more left parenthesis. The rightmost derivation, $E' \Rightarrow_{rm}^* F \Rightarrow_{rm} (E) \Rightarrow_{rm}^* (T) \Rightarrow_{rm} (F)$, can be interpreted as, $(\mid F$, where the viable prefix $'('$ has been seen and F is expected to appear, again showing that $T \rightarrow \bullet F$ is a valid item for this state. On similar lines, the derivation, $E' \Rightarrow_{rm}^* (F) \Rightarrow_{rm} ((E)) \Rightarrow_{rm}^* ((T))$, shows that $T \rightarrow \bullet F$ is a valid item for the viable prefix, $((('$ and also for 3 or more consecutive left parantheses.

You should now be in a position to find answers to both the following problems.

- Given a LR(0) item, say $A \rightarrow \alpha 1 \bullet \alpha 2$ write a regular expression for all the viable prefixes for which this item is valid.
- Given a viable prefix, say β , write all the LR(0) items that are valid for β .

THEOREM : The set of all valid items for a viable prefix β is exactly the set of items reached from the start state along a path labeled β in the DFA that can be constructed from the canonical collection of sets of items, with the transitions given by goto.

- The theorem stated without proof above is a key result in LR Parsing. It provides the basis for the correctness of the construction process we have learnt so far.
- An LR parser does not scan the entire stack to determine when and which handle appears on top of stack (compare with shift reduce parser).
- The state symbol on top of stack provides all the information that is present in the stack. The parser moves from one state s_1 to another state s_2 by shifting a terminal t , because it knows that there is at least one handle which includes t in state s_2 , and the complete handle may get exposed subsequently.
- In a state which contains a complete item, that is a handle, a reduction is called for. However, the lookahead symbols for which the reduction should be applied is not obvious. SLR(1) parser uses the FOLLOW information to guide reductions.

Canonical LR(1) Parser

Canonical LR(1) parser is the next parser that we study. In order to motivate the need for having another LR parser, the limitations of SLR(1) parser need to be explained. As we shall see in the following, the FOLLOW information is imprecise and is the reason for issues with SLR(1) parser.

An example to illustrate this fact is the following grammar, which can be seen to be SLR(1) unambiguous. We shall refer to this grammar as a pointer grammar which generates strings such as :

id *id id = id *id = id id = *id *id = *id **id = *id

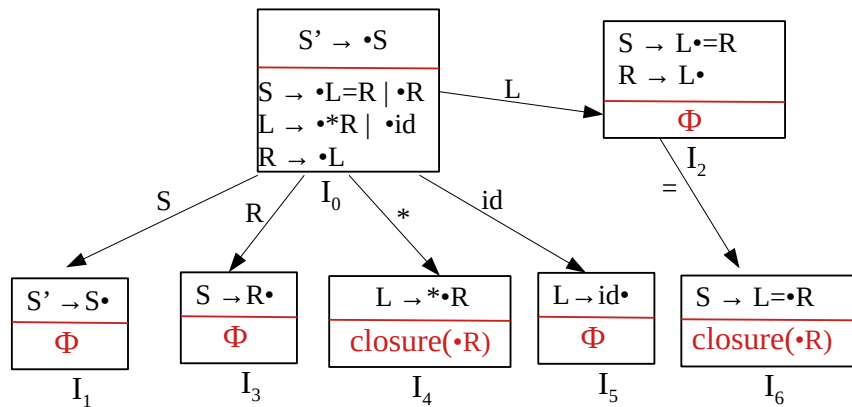
$G = (\{=, *, id\}, \{S, L, R\}, P, S)$ where P , the set of production rules, is given below.

1) $S' \rightarrow S$ (Augmented rule)

2,3) $S \rightarrow L = R \mid R$

4, 5) $L \rightarrow *R \mid id$

6) $R \rightarrow L$



- Consider the state with the viable prefix 'L', labeled as I_2 , as shown above with the LR(0) items present in it.
- The item, $R \rightarrow L \bullet$, in I_2 , calls for reduction by rule 6, that is $R \rightarrow L$. The reduction is called for all symbols in $\text{FOLLOW}(R)$ which is $\{= \$\}$
- The item, $S \rightarrow L \bullet = R$, in I_2 , results in a shifting '=' onto the stack and go to ($I_2, =$) which is numbered, say, as I_6 . The kernel of I_6 is the item $\{S \rightarrow L = \bullet R\}$; other items will get added to I_6 by closure $(\bullet R)$. The summary is the shift move to state 6 on terminal '='.
- The SLR(1) table at this stage of construction is shown below.

State	Action				Goto		
	id	*	=	\$	S	L	R
0	s5	s4	-	-	1	2	3
1	-	-	-	acc	-	-	-
2	-	-	s6 r6	r6	-	-	-

- Action[2, =] shows a shift-reduce conflict. As a consequence the given grammar cannot be directly used by a SLR(1) parser.

SLR(1) Parsing Table for Pointer Grammar

State	Action				Goto		
	id	*	=	\$	S	L	R
0	s5	s4	-	-	1	2	3
1	-	-	-	acc	-	-	-
2	-	-	s6 r6	r6	-	-	-
3	-	-	-	r3	-	-	-
4	s5	s4	-	-	-	8	7
5	-	-	r5	r5	-	-	-
6	s5	s4	-	-	-	8	9
7	-	-	r4	r4	-	-	-
8	-	-	r6	r6	-	-	-
9	-	-	-	r2	-	-	-

You may draw the complete SLR(1) automaton from the parsing table given above by using the same state numbers for the goto() states. As can be seen from the table, except for Action[2, =], there are no other conflicts in any other state.

- The shift move is justified in state 2, because in the new state 6, for the item $S \rightarrow L \bullet = R$, the bullet (\bullet) goes past “=” to the item $S \rightarrow L = \bullet R$ which is a valid item for the viable prefix “L=”
- What about the reduction by $R \rightarrow L \bullet$ in state 2? Can the validity of this move be ascertained like the shift move above?
- The reduce move uses the symbols in FOLLOW(R) and since FOLLOW(R) has { $\$, =$ }, the reduce is also unavoidable.
- The only source of error can be in the use of FOLLOW(). Let the correctness of FOLLOW(R) be examined critically. Can “=” follow the nonterminal R (and NOT *R) in some rightmost derivation ?

$$S' \Rightarrow_{rm} R \Rightarrow_{rm} L \quad (6)$$

$$S' \Rightarrow_{rm} R \Rightarrow_{rm} L \Rightarrow_{rm} *R \Rightarrow_{rm} *L \quad (7)$$

$$S' \Rightarrow_{rm} L = R \Rightarrow_{rm} L = L \Rightarrow_{rm} L = *R \Rightarrow_{rm} L = *L \Rightarrow_{rm} L = *id \Rightarrow_{rm} *R = *id \Rightarrow_{rm} *L = *id \quad (8)$$

- You may examine more rightmost derivations starting from S' to consolidate the observation. In derivations such as (6) and (7), “=” does not follow R in any right sentential form, after L is reduced to R. The candidate L has been highlighted for ease of application.

- However in (8), the last sentential form has “=” following “L”. In this situation when L is reduced to R, we find that “=” follows “*R”. Note that the viable prefix for this state is “*L” which is represented by state 8 in the SLR(1) automaton and not “L” which corresponds to state 2.

Q. What is wrong with FOLLOW ?

The information that “=” belongs to FOLLOW(R) is not correct for state 2. The rightmost sentential forms that permit “=” to follow R are of the form “*L=...” and are taken care of in the state 8 of the parser (see complete SLR(1) table).

The context in state 2 is different (viable prefix is L), and the use of FOLLOW constrains the parser. In other words, FOLLOW() is a global information for the grammar and may not hold for every state of the parser.

Q. How to circumvent this problem ?

Given an item and a state the need is to identify the terminal symbols that can actually follow the lhs nonterminal associated with the item.

An item of the form, $A \rightarrow \alpha \cdot \beta, a$ where the first component is a LR(0) item and the second component is a set of terminal symbols is called a **LR(1) item**. We are using the comma (,) symbol to separate the two components. The second component is known as lookahead symbol. The value 1 indicates that the **length of lookahead is 1**.

- The lookahead symbol is used for a reduce operation only. For an item of the form, $A \rightarrow \alpha \cdot \beta, a$ if β is not ϵ , the lookahead has no effect. But for an item, $A \rightarrow \alpha \cdot, a$ the reduction is applied only if the next token is a.
- It can be shown the terminal set X associated in the 2nd component, $A \rightarrow \alpha \cdot, X$ always satisfies the relation $X \subseteq \text{FOLLOW}(A)$.
- A parser that constructs collection of sets of LR(1) items to build a automaton and a parsing table is called as a Canonical LR(1) parser, also referred to as CLR(1). The precision of the CLR(1) parser is better than SLR(1) parser when $X \subset \text{FOLLOW}(A)$.

VALID LR(1) ITEM

An LR(1) item $A \rightarrow \alpha \cdot \beta, a$ is valid for a viable prefix γ , if there is a derivation

$$S \Rightarrow^*_{\text{rm}} \delta A w \Rightarrow_{\text{rm}} \delta \alpha \beta w, \text{ where}$$

- $\gamma = \delta \alpha$, and
- either a is the first symbol of w, or w is ϵ and a is \$

LR(1) sets of items collection

The basic procedure is same as for LR(0) sets of items collection, the only difference is in specifying how to compute the lookahead for the new LR(1) item obtained by closure.

- If $A \rightarrow \alpha \cdot B\beta, a$ is a valid item for a viable prefix γ , then the item $B \rightarrow \cdot \eta, b$ is also valid for the same prefix γ . Here $B \rightarrow \eta$ is a rule and $b \in \text{FIRST}(\beta a)$.

CONSTRUCTING LR(1) SETS OF ITEMS

The algorithm is similar to the Algorithm for LR(0) sets of item construction and given in the following.

```
procedure items( $G'$ )  
begin  
     $C = \{\text{closure} (S' \rightarrow \cdot S, \$)\};$   
    repeat  
        for each set of items  $I$  in  $C$  and each grammar symbol  $X$ , such that  
             $\text{goto}(I, X)$  is not empty and is not already in  $C$ ,  
        do add  $\text{goto}(I, X)$  to  $C$   
    until no more sets of items can be added to  $C$   
end
```

```
function closure( $I$ )  
begin  
    repeat  
        for each item  $A \rightarrow \alpha \cdot B\beta, a$  and each rule  $B \rightarrow \gamma$  and each  $b \in \text{FIRST}(\beta a)$   
            such that  $B \rightarrow \cdot \gamma, b$  is not in  $I$   
        do add  $B \rightarrow \cdot \gamma, b$  to  $I$ ;  
    until no more items can be added to  $I$ ;  
    return  $I$ ;  
end
```

```
function goto( $I, X$ )  
begin  
     $\text{gotoitems} := \varnothing;$   
    for each item  $A \rightarrow \alpha \cdot X\beta, a$  in  $I$ , such that  $A \rightarrow \alpha X \cdot \beta, a$  is not in  $\text{gotoitems}$   
    do add  $A \rightarrow \alpha X \cdot \beta, a$  to  $\text{gotoitems}$   
     $\text{goto} := \text{closure}(\text{gotoitems});$   
    return  $\text{goto};$ 
```

end

The application of this algorithm to construct a canonical collection of sets of LR(1) items is illustrated in the following.

1. Consider the pointer grammar again. The first collection, i. e., state I_0 , is given by $\text{closure}(S' \rightarrow \bullet S, \$)$. This causes the LR(1) items $S \rightarrow \bullet R, \$$ and $S \rightarrow \bullet L = R, \$$ to be added to I_0 . $\text{Closure}(S \rightarrow \bullet R, \$)$ causes the addition of the item $R \rightarrow \bullet L, \$$ whose closure in turn adds the items $L \rightarrow \bullet *R, \$$ and the item $L \rightarrow \bullet \text{id}, \$$ to I_0 .
2. Closure of the item $S \rightarrow \bullet L = R, \$$ leads to the inclusion of items $L \rightarrow \bullet *R, =$ and $L \rightarrow \bullet \text{id}, =$. The lookaheads for these items are given by $\text{FIRST}(=R \$)$.
3. Since some LR(1) items added by steps 1 and 2 above have the same core, but different lookaheads, we combine the lookaheads to write them in a compact form, $L \rightarrow \bullet *R, \$ =$ and $L \rightarrow \bullet \text{id}, \$ =$.
4. Collection $I_0 = \{S' \rightarrow \bullet S, \$; S \rightarrow \bullet R, \$; S \rightarrow \bullet L = R, \$; L \rightarrow \bullet *R, \$ = ; L \rightarrow \bullet \text{id}, \$ = ; R \rightarrow \bullet L, \$\}$
5. The goto function on I_0 is defined for the symbols S, L, R, id and *. The resulting states are obtained by taking closures of the kernel items as given below.

$$I_1 = \text{goto}(I_0, S) = \text{closure}(S' \rightarrow S \bullet, \$)$$

$$I_2 = \text{goto}(I_0, L) = \text{closure}(S \rightarrow L \bullet = R, \$; R \rightarrow L \bullet, \$)$$

$$I_3 = \text{goto}(I_0, R) = \text{closure}(S \rightarrow R \bullet, \$)$$

$$I_4 = \text{goto}(I_0, *) = \text{closure}(L \rightarrow * \bullet R, \$ =)$$

$$I_5 = \text{goto}(I_0, \text{id}) = \text{closure}(L \rightarrow \text{id} \bullet, \$ =)$$

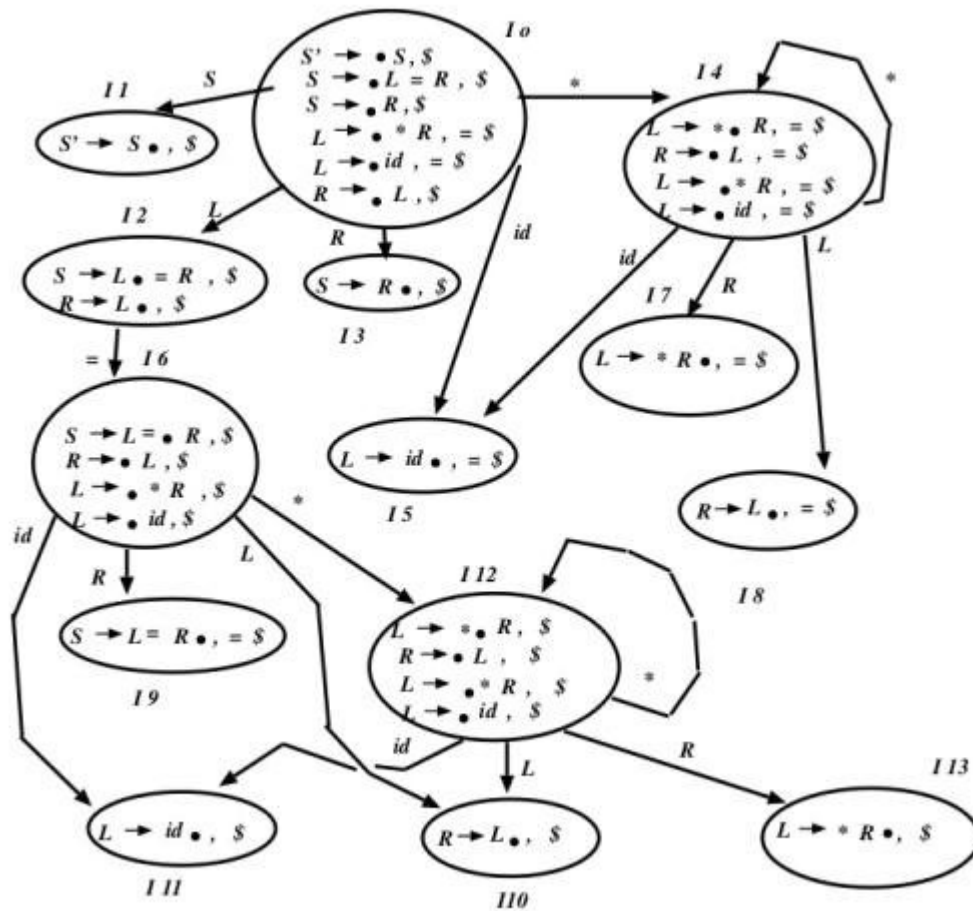
The rest of the collection can be constructed on the same lines. The canonical collection $C = \{I_0, I_1, \dots, I_n\}$ is used to construct the Action and Goto part of the table.

Canonical LR(1) Parsing Table for Pointer Grammar

State	Action				Goto		
	id	*	=	\$	S	L	R
0	s5	s4	-	-	1	2	3
1	-	-	-	acc	-	-	-
2	-	-	s6	r6	-	-	-
3	-	-	-	r3	-	-	-
4	s5	s4	-	-	-	8	7
5	-	-	r5	r5	-	-	-
6	s11	s12	-	-	-	10	9
7	-	-	r4	r4	-	-	-
8	-	-	r6	r6	-	-	-
9	-	-	-	r2	-	-	-
10	-	-	-	r6	-	-	-
11	-	-	-	r5	-	-	-
12	s11	s12	-	-	-	10	13
13	-	-	-	r4	-	-	-

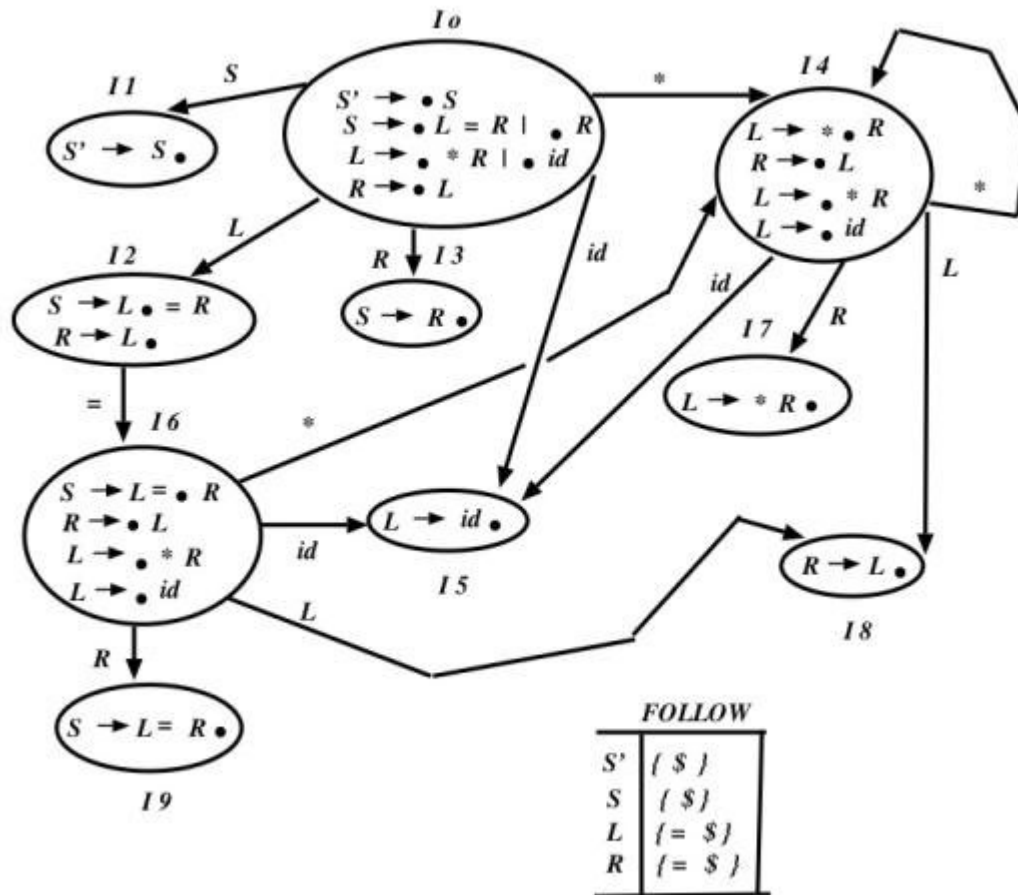
- It is interesting to observe that this automaton has 14 states as compared to SLR(1) automaton which has 10 states.
- The two automaton have identical contents from state 0 to state 5.
- There is reasonable commonality between the remaining states of the two parsers.
- In the parsing table of each parser there are many rows whose entries have high degree of commonality.

CLR(1) Automaton for Pointer Grammar



You may compare the CLR(1) and SLR(1) automaton by keeping them side by side and draw your own conclusions about similarity and differences.

SLR(1) Automaton for Pointer grammar



LALR(1) PARSER CONSTRUCTION

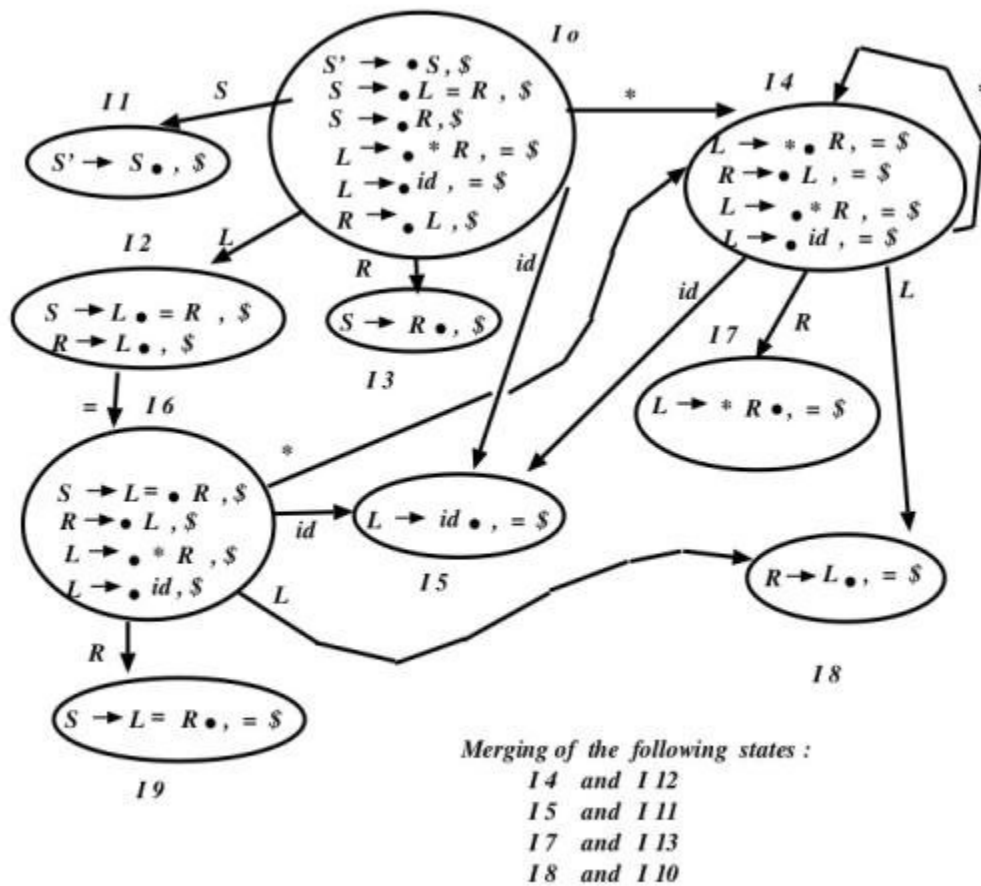
- This parser is of intermediate capability as compared to SLR(1) and CLR(1). The parser has the size advantage of SLR and lookahead capability like CLR.
- In a CLR parser, several states may have the same first components (LR(0) items part), but differ in the lookaheads associated and hence are represented as distinct states.
- In terms of the first component only, both SLR and LR define the same collection of LR(0) items.
- LALR automaton is created by merging states of LR automaton that have the same core (set of first components). The merge results in the union of the lookahead symbols of the respective items.
- This method of LALR(1) automaton construction enforces that CLR(1) automaton be constructed first. However as we show later, one construct LALR(1) automaton directly from the grammar instead of building it via CLR(1) automaton. The reason most literature on LR parsing uses this indirect method is that understanding the essential difference between LALR(1) and CLR(1) is better understood this way.

LALR(1) Parsing Table Construction

1. Construct the collection $C = \{I_0, I_1, \dots, I_n\}$ of CLR(1) items.
2. For each core (set of kernel items) among the set of CLR(1) items, find all sets having the same core and replace these sets by their union.
3. From the reduced collection, say, $C = \{J_0, J_1, \dots, J_m\}$, the Action table is constructed the same way as that for CLR(1).
4. To compute $\text{Goto}[J, X]$, let $J = I_{p1} \cup I_{p2} \cup \dots \cup I_{pr}$, that is collection J of LALR(1) items is the union of all the r collections CLR(1) items, $\{I_{p1}, I_{p2}, \dots, I_{pr}\}$, such that all these CLR(1) collections have the same core. Then construct $K = \{\text{Goto}(I_{p1}, X) \cup \text{Goto}(I_{p2}, X) \cup \dots \cup \text{Goto}(I_{pr}, X)\}$. Finally, we determine, $\text{Goto}[J, X] = K$.

To illustrate the construction of a LALR automaton, consider the LR(1) automaton for the pointer grammar given earlier.

- Examine this automaton to identify the states whose cores are identical. We find that pair of states, given by (I_4, I_{12}) , (I_5, I_{11}) , (I_7, I_{13}) , and (I_8, I_{10}) have identical cores.
- The merger of the states given above results in $I_4 = I_4 \cup I_{12}$, $I_{45} = I_5 \cup I_{11}$, $I_7 = I_7 \cup I_{13}$, and $I_8 = I_8 \cup I_{10}$. This happens since in each case the lookahead sets are proper subsets.
- The LALR automaton and the parsing table for the grammar are given in the following.



You may observe that in I_2 , the shift reduce conflict of SLR(1) has vanished, exactly the same way it was absent in the corresponding state of CLR(1).

LALR(1) Parsing Table for pointer grammar follows from the LALR(1) automaton constructed above.

LALR(1) Parsing Table for Pointer Grammar

State	Action				Goto		
	id	*	=	\$	S	L	R
0	s5	s4	-	-	1	2	3
1	-	-	-	acc	-	-	-
2	-	-	s6	r6	-	-	-
3	-	-	-	r3	-	-	-
4	s5	s4	-	-	-	8	7
5	-	-	r5	r5	-	-	-
6	s5	s4	-	-	-	8	9
7	-	-	r4	r4	-	-	-
8	-	-	r6	r6	-	-	-
9	-	-	-	r2	-	-	-

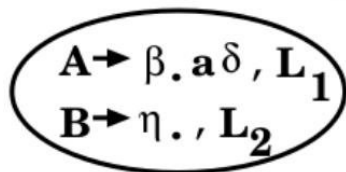
LALR(1) GRAMMAR AND PARSER

A grammar for which there is a conflict free LALR(1) parsing table is called a LALR(1) grammar and a parser which uses such a table is known as LALR(1) parser.

RESULTS IN LALR(1) PARSING

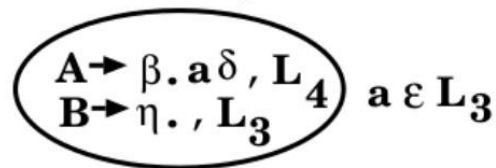
1. A theorem states that a LALR(1) parsing table is always free of shift reduce conflicts, provided the corresponding LR(1) table is so.
2. The key observation towards proving this fact is given in the following figure.

Merged State I_{ij}



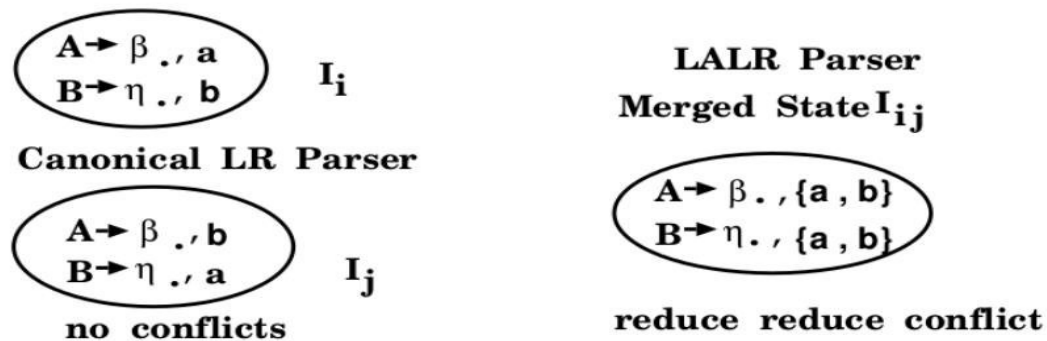
LALR Parser
 $a \in L_2$
shift reduce conflict

**One of the original states
 I_i or I_j**



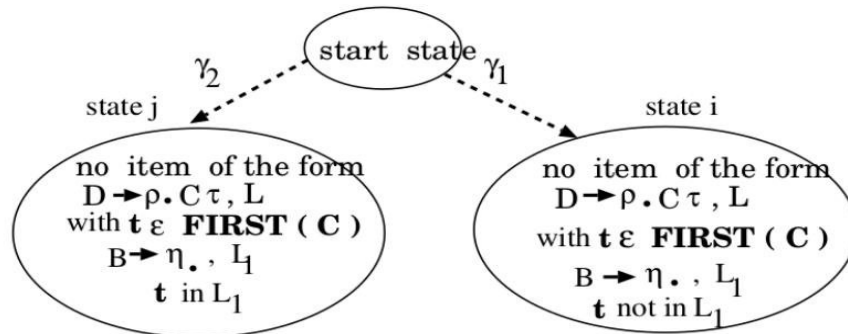
Canonical LR Parser
shift reduce conflict

3. However the theorem does not exclude the possibility of a LALR(1) parsing table having reduce reduce conflicts while the corresponding LR(1) table is conflict free as shown in the situation below.

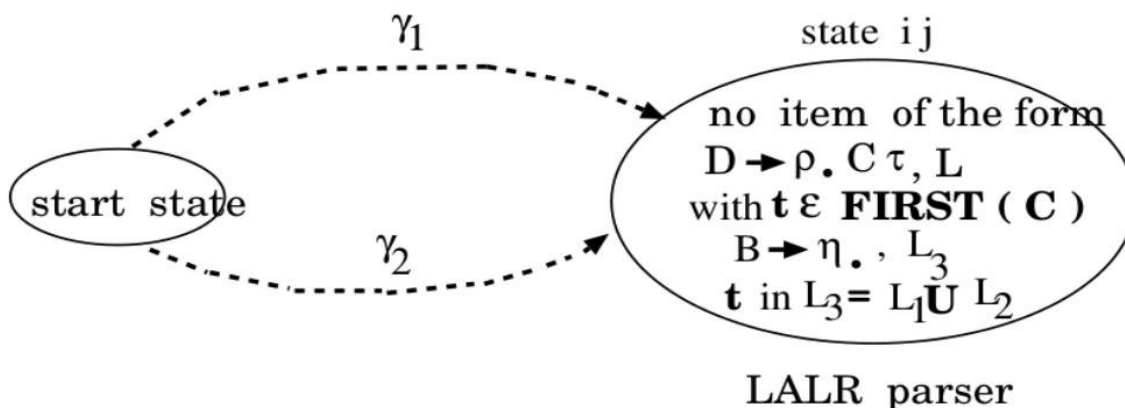


4. Another important result relates the behaviour of these two parsers. It can be formally established that the behaviour of LALR and LR parsers for the same grammar G are identical, that is the same sequence of shifts and reduces, for **a sentence** of the language, $L(G)$.
5. However, for an erroneous input, a LALR parser may continue to perform reductions after the LR parser would have caught and reported the error. An interesting fact is that a LALR parser **does not shift any symbol beyond the point** that the LR parser declared an error.
6. Use the parsers of LALR(1) and LR(1) to parse the input, $id = *id = \$$, and find the difference in behaviour. The reasons are analyzed in the following. The discussion can be converted into a proof for this result if LALR has only one merged state.

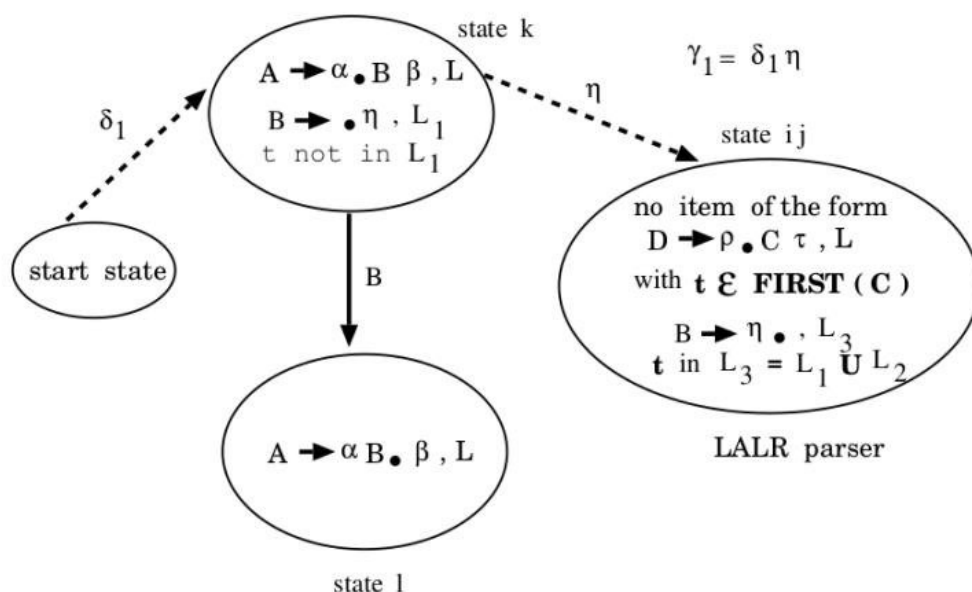
7. Let a CLR(1) parser be in state i with t as the next symbol in the input. This state i recognizes a viable prefix γ_1 . In the figure given below, state i indicates error on t since the first form of items, $D \rightarrow \rho \cdot C \tau, L$ where $t \notin \text{FIRST}(C)$, do not permit t to be shifted. The other LR(1) items of the form, $B \rightarrow \eta \cdot, L_1$ with $t \notin L_1$ prevent any reduce on t . CLR(1) parser will report an error in state i with t as the input symbol. However in the state j of the CLR(1) automaton, which has the same core as that of state i , while a shift on t is not permitted, a reduce action is possible on t .



8. State ij of the LALR(1) automaton, which is the result of merger of states i and j of the corresponding CLR(1) parser, is shown below. This state after recognizing γ_1 permits a reduce on t because of the state j of the LR parser. State ij permits a reduce action on t , inherited from state j of CLR(1).



9. On input t , state ij of LALR(1) will execute reduce by $B \rightarrow \eta \bullet$, and the LALR parser would reach state l whose relevant contents are shown below. It is assumed that states k and l of both the parsers are identical.



Presence of the item $A \rightarrow \alpha \bullet B \beta, L$ in state k is justified since $B \rightarrow \bullet \eta, L_1$ is an item in state k , and can only be added through closure. Terminal t must not be in $\text{FIRST}(\beta)$, since otherwise $t \in L_1$ in state i of the CLR(1) automaton, which is given to be false. Therefore, since $t \notin \text{FIRST}(\beta)$ in state k , this fact is also true for state l whereby state l does not permit a shift on t and consequently detects and reports the error in state l of the LALR(1) parser. This is different from the behaviour of CLR(1) parser which had reported the error in state i . The corresponding LALR(1) parser performs an extra reduce and reported the error in a different state l , without consuming the terminal t .

COMPARISON OF THE THREE LR PARSERS

The observations on SLR(1), CLR(1) and LALR(1) parsers are summarized below.

1. SLR parser is the one with the smallest size and easiest to construct. It uses FOLLOW information to guide reductions.
2. Canonical LR parser has the largest size among the other parsers of this family. The lookaheads are associated with the items and they make use of the left context available to the parser.
3. CLR grammar is a larger subclass of context free grammars as compared to that of SLR and LALR grammars.
4. LALR parser has the same size as that of a SLR parser but is applicable to a wider class of grammars than SLR grammar.

5. LALR is less powerful as compared to CLR, however most of the syntactic features of programming languages can be expressed in this grammar.
6. LALR parser for an erroneous input behaves differently than a CLR parser. Error detection capability is immediate in CLR but not so in LALR.
7. Both CLR and LALR parsers are expensive to construct in terms of time and space. However, efficient methods exist for constructing LALR parsers directly.

AUTOMATIC GENERATION OF THE LR PARSERS

All the parsers of this family can be generated automatically given a context free grammar as input. The parser generation tool YACC available in UNIX uses such a strategy for generating a LALR parser.

The basic steps to automate the process for generating a SLR parser from a grammar G are given below.

1. Augment the given grammar G .
2. Construct FOLLOW information for all the nonterminals of G .
3. Use the algorithms for Closure , goto and sets-of-items construction to construct the canonical collection of LR(0) sets of items.
4. Use the procedure for constructing a parsing table from the canonical collection to create the table. If the table contains conflicts, report them.
5. Supply a driver routine which uses a stack, an input buffer and the parsing table for actually parsing an input.

The other parsers can also be generated along similar lines.

**STATIC SEMANTICS
&
INTERMEDIATE CODE GENERATION**

Supratim Biswas

Course Material

March 2024

Department of Computer Science & Engineering
BIT Mesra

Principles of Compiler Design : Semantic Analysis and Intermediate Code Generation

Theme 1 : Semantic analysis - concepts and examples; Syntax Directed Translation Scheme (SDTS); semantic analysis of declarations; semantic analysis of expressions in C/C++. Intermediate code forms. Illustration through examples.

Theme 2 : Semantic analysis of assignment statement; translation of boolean expressions - partial and complete evaluation; translation of control flow statements. Illustration through examples.

Theme 3 : Translation of procedure calls; runtime environments and activation records. Illustration through examples.

Pre-requisite : This module assumes familiarity with scanning and parsing, particularly LR parsers.

Text Book :

Compilers – Principles, Techniques and Tools, Alfred V Aho, Monica S Lam, Ravi Sethi and Jeffrey D Ullman, Pearson Education Inc, 2nd Edition, 2007

Material of this course is covered in Chapters 5, 6 and 7, pages 303 to 502 of this book.

Syntax Analysis vs Semantic Analysis

Which of the following situation falls under the scope of semantic analysis ?

- The compiler reports errors but how to know whether the error was detected during lexical phase or syntax phase or beyond these two phases.
- A program is syntactically correct but the compiler reports errors. It implies that the parser alone would have passed the program as a valid string in $L(G)$. However the string has errors in the sense that it can not be assigned unique meaning. It should be possible to create many instances of such errors.
- A program compiles correctly : implies that there are no compilation errors reported. However when the program is executed, it is terminated abnormally with run-time errors. The behavior of the program at run-time is not as expected by the programmer.

- A program compiles, executes and produces output. But the output is not the desired one.

We would examine several programs in C++ to get some insight into the issues mentioned above and also to motivate us to learn semantics.

The programs will be compiled through g++ and our intuitions will be matched with the actions performed by g++.

Example Program 1	Example Program 2	Example Program 3
<pre>int main() { int a = b; int b = 5; return 0; }</pre>	<pre>int main() { int a = b, b = 5; return 0; }</pre>	<pre>int main() { float b; int b = 5; return 0; }</pre>

Program 1 : An error message is given by g++.

p1.C: In function :

p1.C:6:11: error: was not declared in this scope

```
6 | { int a = b;
  |           ^
```

Program 2 : Error message

p2.C: In function :

p2.C:6:11: error: was not declared in this scope

```
6 | { int a = b; b = 5;
  |           ^
```

Program 3 : Error message

p3.C: In function :

p3.C:7:7: error: conflicting declaration

```
7 | int b = 5;
  |      ^
```

p3.C:6:9: note: previous declaration as

```
6 | { float b;
  |      ^
```

Example Program 4	Example Program 5	Example Program 6
<pre>int main() { int &i; cout << i << endl; return 0; }</pre>	<pre>int main() { short s = 1234567890; cout << s << endl; return 0; }</pre>	<pre>int main() {int i = 40; if (1 <= i <= 5) cout << " In range \n" ; else cout << " Out of Range \n"; return 0; }</pre>

Program 4 : Error message

p4.C: In function :

p4.C:6:8: error: declared as reference but not initialized

```
6 | { int &i;
  |      ^
```

Program 5 : Warning Message

p5.C: In function :

p5.C:6:13: warning: overflow in conversion from to changes value from to [-Woverflow]

```
6 | { short s = 1234567890;
  |      ^~~~~~
```

Program 6 : No compilation error.

Produces the following output on execution :

In range

Example Program 7	Example Program 8	Example Program 9
<pre>int main() { int a[5] = {1, 2, 3, 4, 5, 6, 7, 8}; return 0; }</pre>	<pre>int main() { int a[5] = {1, 2, 3}; int sum; for (int i = 0; i < 10000; i++) sum = sum + a[i]; cout << sum << endl; return 0; }</pre>	<pre>int f(int x) { if (x > 10) return x; else if (x > 5) return x+5; } int main() { int i = -5; cout << f(i) << endl; return 0; }</pre>

Program 7 : Error Message

p7.C: In function :

p7.C:6:37: error: too many initializers for

```
6 | { int a[5] = {1, 2, 3, 4, 5, 6, 7, 8};
  |                                     ^
```

Program 8 : No Compilation error.

In execution, produces the following message.

Segmentation fault (core dumped)

Program 9 : Warning message.

p9.C: In function :

p9.C:8:1: warning: control reaches end of non-void function [-Wreturn-type]

```
8 | }
  | ^
```

Example Program 10	Example Program 11	Example Program 12
<pre>int main() { float inc = 0.01;</pre>	<pre>/* * Test Program</pre>	<pre>short f(short a) { cout << " short \n";</pre>

Example Program 10	Example Program 11	Example Program 12
<pre>float sum = 0.0; while (inc != 0.1) { cout << inc << endl; inc = inc + 0.01; } cout << sum << endl; return 0; }</pre>	<pre>* int main() { int a = b; int b = 5; return 0; }</pre>	<pre>return a;} long f(long x) { cout << "long \n"; return x;} char f (char c) { cout << "char \n"; return c;} int main() { f(100);}</pre>

Program 10 : No Compilation error.

On execution, no output is received.

Program 11 : Error message.

p11.C:4:1: error: unterminated comment

```
4 | /*
  | ^
```

Program 12 : Error message.

p12.C: In function :

p12.C:10:7: error: call of overloaded is ambiguous

```
10 | f(100);
    | ^
```

p12.C:3:7: note: candidate:

```
3 | short f(short a) { cout << " short \n"; return a;}
    | ^
```

p12.C:4:6: note: candidate:

```
4 | long f(long x) { cout << " long \n"; return x;}
    | ^
```

p12.C:5:6: note: candidate:

```
5 | char f (char c)
    | ^
```

Example Program 13
<pre>long f(long a) { cout << " long \n"; return a;} int f(int x) { cout << " int \n"; return x;} char f (char c) { cout << " char \n"; return c;} int main() { short d = 25;</pre>

Example Program 13

```
char ch = '$';  
f(1000000000000000);    f(1234);  f(ch);    f(d);  
}
```

Program 13 : No compilation error.

On execution, the following output is produced.

```
long  
int  
char  
int
```

Take Home Exercise : Write different instances of syntactically valid programs in C/C++ that are not semantically acceptable.

Summary :

- The programs reveal the behavior of the g++ compiler through different program examples.
- You have been exposed to a major part of the front-end of a compiler that comprises of Scanning (Lexical analysis) and Parsing (Syntax Analysis).
- The semantics analysis module of a compiler logically starts after parsing but in practice it is

interleaved with syntax analysis. Of course, this is not mandatory and semantic analysis could have been defined by a separate pass (or passes) post syntax analysis also but as we shall see the first approach turns out to be efficient and adequate.

- The semantic analysis that we shall discuss in this module is based on a bottom-up parser and we will assume an LALR(1) parser for our purpose. This does not preclude performing semantic analysis with top-down parsers but that is not under this course.

The first two phases that you have learnt so far can be used in applications, other than compilers, as well. The characteristic of such an application should involve recognizing strings from a language that is generated by a context free grammar.

The subsequent phases of the compiler that we shall study are closely tied to implementation of programming language features. Instead of dealing with all the semantic related issues for a specific PL, in this course we take the stand that interesting linguistic features across PL be examined.

The following approach would be adopted for the semantic analysis part of this course.

- Abstract a linguistic feature for examination and determine from language reference manual / compiler writers guidelines, the meaning specified for the feature.
- At the simplest level of abstraction, we shall often use English as our description mechanism. However it should be obvious that a formal notation for specifying semantics is more desirable for precise description and unambiguous interpretation.
- A Context Free Grammar (CFG), G that generates all the syntactically correct strings of the underlying language $L(G)$, is to be chosen among various equivalent alternatives. However the set of all syntactically correct strings may include several strings that are not necessarily semantically valid. WHY ?
- The choice of the grammar rules are dictated by the semantic actions that determine whether the syntactically valid $w \in L(G)$ is also semantically correct. Knowing that the string w is syntactically valid is not enough for semantic analysis.

Review of Background in Compilers

The front-end of a compiler comprises of three phases :

- Lexical analysis or scanning
- Syntax analysis or parsing
- Semantic analysis and intermediate code generation

We assume familiarity with the first two phases. These two phases can be used in applications, other than compilers, as well. The characteristic of such an application should involve recognizing strings from a language that is generated by a context free grammar.

The subsequent phases of the compiler that we shall study are closely tied to implementation of programming language features. Instead of dealing with all the semantic related issues for a specific PL, this course advocates that interesting linguistic features across PL be examined.

The following approach would be adopted for the semantic analysis part of this course.

- Abstract a linguistic feature for examination and determine from language reference manual / compiler writers guidelines, the meaning specified for the feature.
- At the simplest level of abstraction, we shall often use English as our description mechanism. However it should be obvious that a formal notation for specifying semantics is more desirable for precise description and unambiguous interpretation.
- A CFG, G that generates all the semantically correct strings of the underlying language $L(G)$, including others that are not, is chosen among various equivalent grammars.
- The choice of the grammar rules are dictated by the semantic actions that determine whether the syntactically valid $w \in L(G)$ is also semantically correct.
- Knowing that the string w is semantically valid is a necessary requirement for a compiler but not sufficient.
- Besides semantic validation, a simpler equivalent representation of the source is generated in a format known as intermediate code.
- In summary the purpose of semantic analysis is to find the semantics of the source code and

express it in the intermediate code instructions such that the semantics of both remain the same.

- An important characteristics of intermediate code is that generation of assembly/machine code on the target computer is a relatively much simpler problem than generating target code directly from the source program.

BASIC CONCEPTS AND ISSUES

WHAT IS STATIC SEMANTIC ANALYSIS?

Static semantic analysis ensures that a program is correct, in that it conforms to certain extra-syntactic rules.

The analysis is called

Static: because it can be done by examining the program text, and

Semantic: because the properties analysed are beyond syntax, i.e. they cannot be captured by context free grammars.

Examples of such analyses are:

- a. Type analysis,**
- b. Name and scope analysis,**
- c. Processing of declarations**
- d. Processing of expressions and statements**
- e. Intermediate code generation.**

TYPE ANALYSIS

The type of an object should match that expected by its context.

Examples of issues involved in type analysis are:

1. Is the variable on the left hand side of an assignment statement 'compatible' with the expression on the right hand side?
2. Is the number of actual parameters and their types in a procedure call compatible with the formal parameters of the procedure declaration?
3. Are the operands of a built-in operator of the right type.

If not, can they be forced (coerced) to be so?

What is the resultant operator, then, to be interpreted as?

For example, the expression **3 + 4.5** can be interpreted as:

- a. Integer addition of 3 and 4 (4.5 truncated).
- b. Real addition of 3.0 (3 coerced to real) and 4.5.

Note: The notion of 'type compatibility' is non-trivial. For example, in the declaration

type ptr = ↑ object;

var a : ↑ object; b : ptr;

are a and b type compatible?

To handle the issues related to type compatibility, two notions of type equivalence were introduced by PL designers.

- Structural Equivalence
- Name Equivalence

Before analysing a program in a given L for type equivalence, one must find out the type equivalence adopted by the designers of L.

NAME AND SCOPE ANALYSIS

The issues are:

1. What names are visible at a program point? Which declarations are associated with these names?
2. Conversely, from which regions of a program is a name in a declaration visible? Example :

DECLARATION PROCESSING

Declaration processing involves:

- 1) *Uniqueness checks*: An identifier must be declared uniquely in an declaration.
- 2) *Symbol Table updation*: As declarations provide most of the information regarding the attributes of a name, this information must be recorded in a data structure called the symbol table. The main issues here are:
 - a) What information must be entered into the symbol table, and how should such information be represented?
 - b) What should be the organization of the symbol table itself.
- 3) *Address Resolution*: As declarations are processed, the addresses of variables may also be computed. This address is in the form of an offset relative to the beginning of the storage allocated for a block (refer to the module on runtime environments for more details).

INTERMEDIATE CODE GENERATION

The front end translates the program into an intermediate representation, from which the back end generates the target code.

There is a choice of intermediate representations such as abstract syntax trees, postfix code and three address code, Gimple among many others.

EXAMPLE

Three address code generated for the assignment statement $x := A[y,z] < b$, where A is a 10 x 20 array of integers, and b and x are reals is:

1. $t1 := y * 20$	6. if $t4 < b$ goto 9
2. $t1 := t1 + z$	7. $t5 := 0$
3. $t2 := \text{addr}(A) - 21$	8. goto 10
4. $t3 := t2[t1]$	9. $t5 := 1$
5. $t4 := \text{intoreal}(t3)$	10. $x := t5$

SYNTAX DIRECTED ANALYSIS

Can we always represent the properties stated above through a context free grammar? There are theoretical results to show that the answer is in the negative.

1. Declaration of a variable before its use is a requirement in most programming languages. The formal language $\{w c w \mid w \text{ is a string from some alphabet } \Sigma\}$ can be thought of as an abstraction of languages with a single identifier declaration (the first occurrence of w), which must be declared before use (the second occurrence of w).

This language is not context free.

2. The language $\{a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1\}$ is an abstraction of languages with two procedures and their corresponding calls, which require that the number of formal parameters (represented by the a 's and b 's) be the same as the number of actual parameters (c 's and d 's).

This language is not context free.

However, semantic analysis has traditionally been done in a *syntax directed* fashion, i. e., along with parsing, Why?

SYNTAX DIRECTED DEFINITION

Associate with each terminal and non-terminal a set of values called *attributes*. The evaluation of attributes takes place during parsing, i. e., when we use a production of the form, $A \rightarrow X Y Z$, for *derivation* or *reduction*, the attributes of X , Y and Z could be used to calculate the attributes of A .

EXAMPLE: For the grammar shown below, suppose we wanted to generate intermediate code:

$$S \rightarrow \text{id} := E$$
$$E \rightarrow E_1 + E_2$$
$$E \rightarrow E_1 * E_2$$
$$E \rightarrow - E_1$$
$$E \rightarrow \text{id}$$

It is convenient to have the following attributes:

S.code - The intermediate code associated with S .

E.code - The intermediate code associated with E .

E.place - The temporary variable which holds the value of E.

id.place - The lexeme corresponding to the token id.

The grammar is augmented with the following semantic rules:

$S \rightarrow id := E$

$S.code := E.code \parallel gen(id.place ':=' E.place);$

$E \rightarrow E_1 + E_2$

$E.place := newtemp;$

$E.code := E_1.code \parallel E_2.code \parallel$

$gen(E.place ':=' E_1.place '+' E_2.place);$

$E \rightarrow E_1 * E_2$

$E.place := newtemp;$

$E.code := E_1.code \parallel E_2.code \parallel$

$gen(E.place ':=' E_1.place '*' E_2.place);$

$E \rightarrow - E_1$

$E.place := newtemp;$

$E.code := E_1.code \parallel$

$gen(E.place ':=' 'uminus' E_1.place);$

$E \rightarrow id$

$E.place := id.place; E.code := '';$

where

- 1) newtemp is a function which generates a new temporary variable name, each time it is invoked.
- 2) \parallel is the concatenation operator.
- 3) $gen()$ evaluates its non-quoted arguments, concatenates the arguments in order, and returns the resulting string.

4) Symbols in quotes are inserted in place after stripping off the quotes.

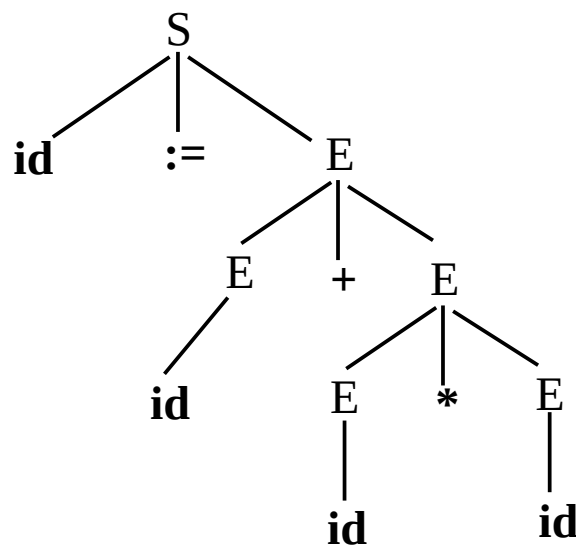
The semantic rule

$S.code := E.code \parallel \text{gen}(id.place := E.place);$

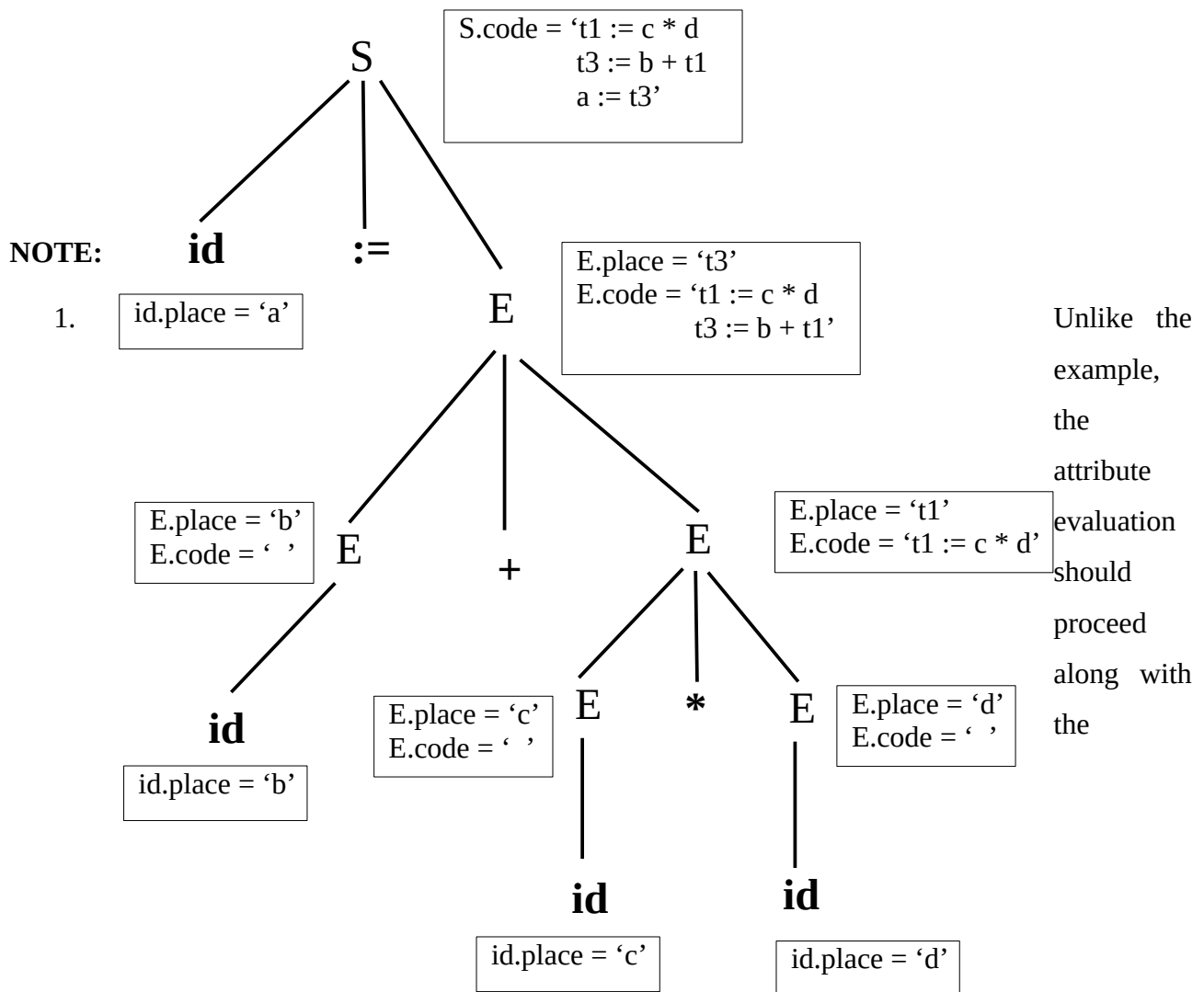
associated with the rule $S \rightarrow id := E$ is interpreted as :

- When the the right hand of the rule, $id := E$, is seen by a bottom up parser, the semantic attributes of the symbols in the rhs are assumed to be known.
- The semantic attributes, $id.place$ and $E.code$, are available with the parser.
- After reducing the rhs, $id := E$, by its lhs nonterminal S , the parser appends to the attribute $E.code$ another code fragment expressed as $id.place := E.place$
- The physical interpretation is that after the code for E has been generated in $E.code$, the result of $:=$ is to assign to $id.place$ the value that is currently held in $E.place$.

Consider the sentence, $a := b + c * d$ generated by the assignment grammar and its parse tree shown below.



SYNTAX DIRECTED ANALYSIS



construction of the parse tree. This would introduce restrictions on the possible relations between attribute values.

2. Nothing has been said about the implementational details of attribute evaluation.
3. The attributes of terminals are usually supplied by the lexical analyser.

SYNTAX DIRECTED DEFINITION: FORMALIZATION

A syntax directed definition is an **augmented context free grammar**, where each production $A \rightarrow \alpha$, is associated with a set of semantic rules of the form

$b := f(c_1, c_2, \dots, c_k)$, where f is a function, and

1. b is a *synthesized attribute* of A and c_1, c_2, \dots, c_k are attributes belonging to the grammar symbols of α , or

2. b is an *inherited attribute* of one of the grammar symbols α , and c_1, c_2, \dots, c_k are attributes belonging to A or α .

In either case, we say that the attribute b depends on the attributes c_1, c_2, \dots, c_k .

In the previous example, $S.code$, $E.code$, and $E.place$ were all synthesized attributes.

TRANSLATION SCHEMES

A translation scheme is a refinement of a syntax directed definition in which:

- i. Semantic rules are replaced by actions.
- ii. The exact point in time when the actions are to be executed is specified. This is done by embedding the actions within the right hand side of productions.

In the translation scheme

$$A \rightarrow X_1 X_2 \dots X_i \{ \text{action} \} X_{i+1} \dots X_n$$

action is executed after completion of the parse for X_i .

EXAMPLE : A translation scheme to map infix expressions into postfix expressions. Expressions involve binary *addop* operators $\{+, -\}$ and token **num**.

For the infix expression : $10 + 7 - 3$

Desired translation : $10\ 7\ +\ 3\ -$

Using expression grammar writing rules for attributes of *addop*

$$E \rightarrow E\ \text{addop}\ T \mid T$$

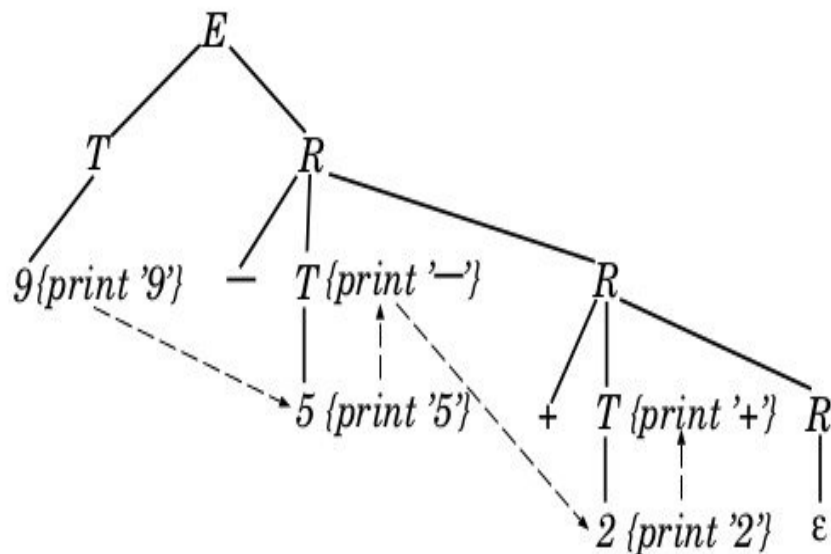
$$T \rightarrow \text{num}$$

We however use an equivalent grammar that works for both parsing methods.

Consider the translation scheme given below.

$$E \rightarrow T R$$

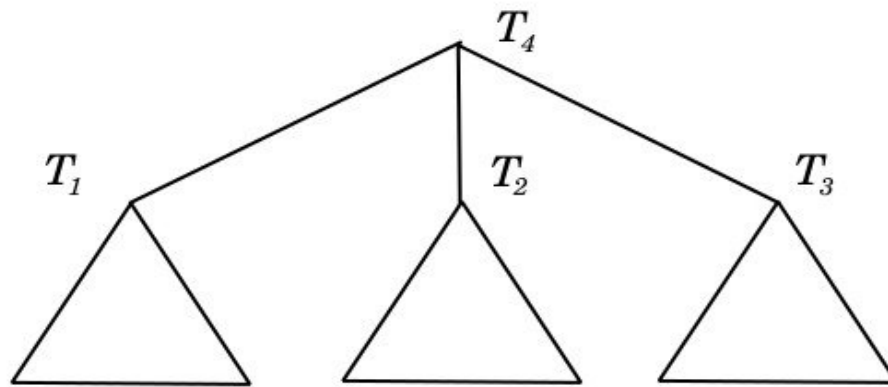
$$R \rightarrow \text{addop } T \{ \text{print}(\text{addop.lexeme}) \} R_1 \mid \epsilon$$

$$T \rightarrow \text{num} \{ \text{print}(\text{num.val}) \}$$


The order of execution of actions for the string $9 - 5 + 2$ is shown by dotted arrows. Examine how this translation scheme works for top-down and bottom-up parsers and also whether it changes the properties of the operators.

RESTRICTIONS ON ATTRIBUTE DEPENDENCES

Certain restrictions must be placed on the dependences between attributes, so that they can be evaluated along with parsing. It is interesting to note that top-down parsers or bottom-up parsers would construct the parse tree shown in the figure in the order T1, T2, T3, T4.



Clearly an attribute of T2 cannot

depend on any attribute of T3.

The extent of restriction gives rise to two classes of syntax directed definitions,

a. **S-attributed definition**

b. **L-attributed definition**

We will not discuss L-attributed definitions in this course.

S-ATTRIBUTED DEFINITIONS

A translation scheme is *S-attributed* if:

- (i) Every non-terminal has synthesized attributes only.
- (ii) All actions occur on the right hand side of productions.

The syntax directed definition shown below is an example of an S-attributed definition:

$S \rightarrow id := E$

$S.code := E.code \parallel gen(id:place := E.place);$

$E \rightarrow E_1 + E_2$

$E.place := newtemp;$

$E.code := E_1.code \parallel E_2.code \parallel$
 $gen(E.place := E_1.place '+' E_2.place);$

$E \rightarrow E_1 * E_2$

$E.place := newtemp;$

$E.code := E_1.code \parallel E_2.code \parallel$
 $gen(E.place := E_1.place '*' E_2.place);$

$E \rightarrow - E_1$

$E.place := newtemp;$

$E.code := E_1.code \parallel$
 $gen(E.place := 'uminus' E_1.place);$

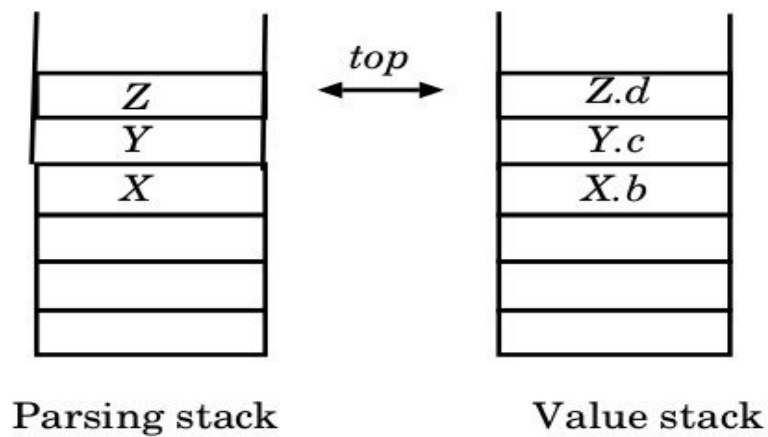
$E \rightarrow id$

$E.place := id.place; E.code := '';$

IMPLEMENTATION OF S-ATTRIBUTED DEFINITION

Assume a bottom up parsing strategy. We augment the parsing stack with a value stack (val). If location i in the parsing stack, $parse[i]$, contains a grammar symbol A , then $val[i]$ will contain all the attributes of A .

Prior to a reduction using a production $A \rightarrow X Y Z$:

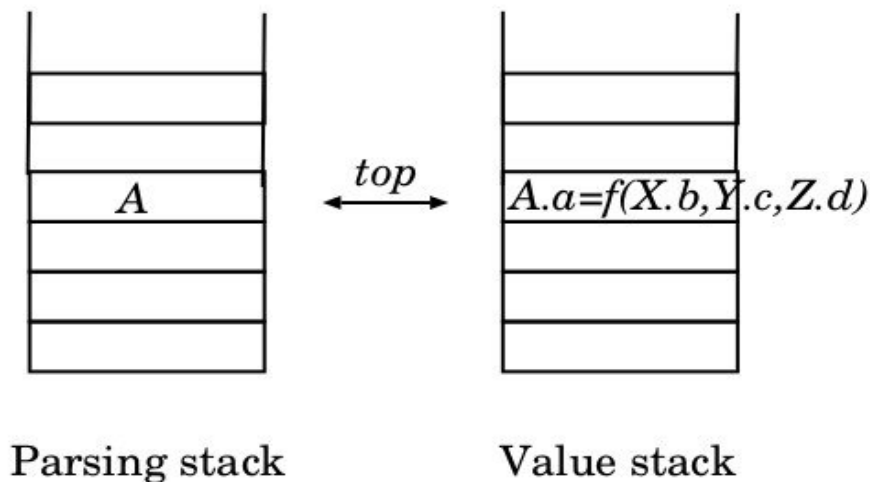


Note that rhs $X Y Z$ is on parse stack in the reverse order.

If an attribute a of lhs nonterminal A is defined as $f(X.b, Y.c, Z.d)$,

$$A.a = f(X.b, Y.c, Z.d)$$

then, after the reduction, the contents of parse and val stacks change as shown.



We can now replace the semantic rules by actions which refer to the actual data structures used to hold and manipulate the attributes. This is illustrated for the assignment grammar discussed earlier.

Let each element of val stack be a record with two fields `val[i].code` and `val[i].place`. Then the syntax directed definition is :

- First grammar rule with semantic rules given earlier

$S \rightarrow id := E$

$S.code := E.code \parallel gen(id.place := E.place);$

The same rule with actions is written as :

$S \rightarrow id := E$

$val[top-2].code := val[top].code \parallel$

$gen(val[top-2].place := val[top].place);$

- The second rule

$E \rightarrow E_1 + E_2$

$E.place := newtemp;$

$E.code := E_1.code \parallel E_2.code \parallel$

$gen(E.place := E_1.place '+' E_2.place);$

is rewritten as

$E \rightarrow E_1 + E_2$

$T := newtemp;$

$val[top-2].code := val[top-2].code \parallel val[top].code \parallel$

$gen(T := val[top-2].place '+' val[top].place);$

$val[top-2].place := T;$

- The action for the rule for $E \rightarrow E_1 * E_2$ is rewritten as

```

T := newtemp;
val[top-2].code := val[top-2].code || val[top].code
|| gen( T ':= ' val[top-2].place) '*'
val[top].place;
val[top-2].place := T;

```

- $E \rightarrow - E_1$

```

T := newtemp;
val[top-1].code := val[top].code ||
gen( T ':= ' 'uminus' val[top].place) '*'
val[top].place;
val[top-1].place := T;

```

- The last rule is rewritten as

```

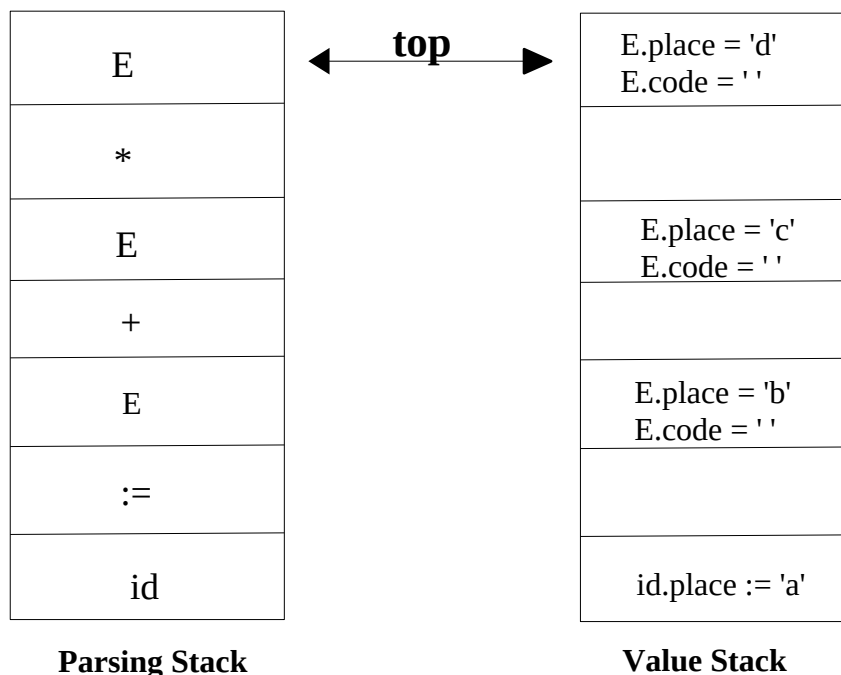
E → id
val[top].place := val[top].place; val[top].code := ' ';

```

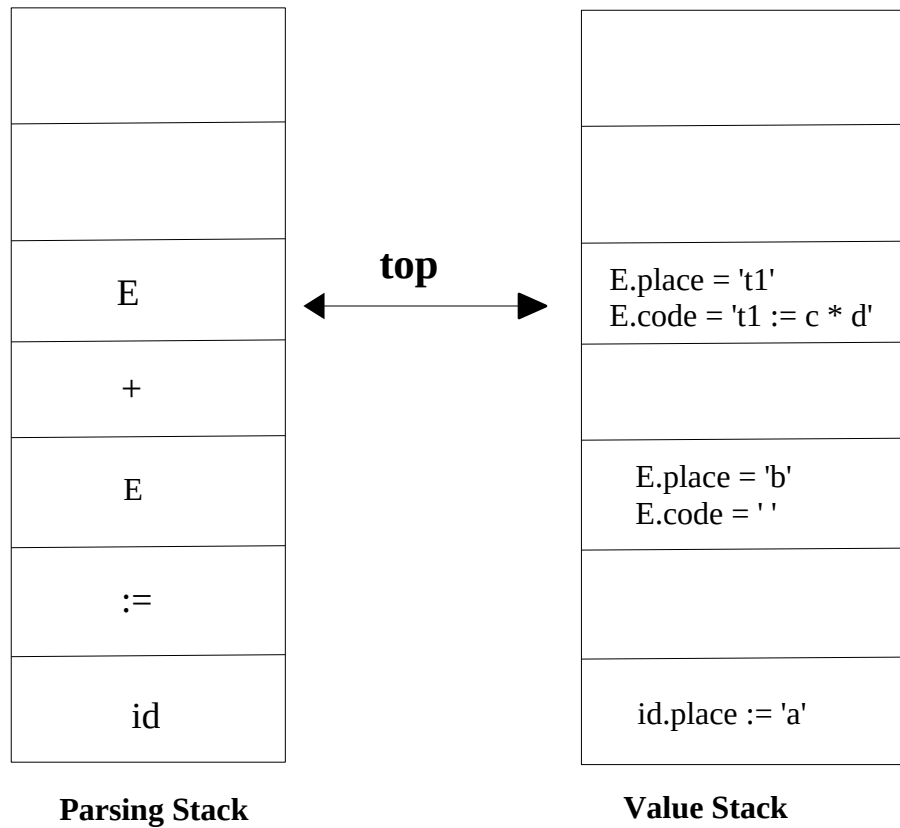
- The illustration shows how semantic rules can be transformed to semantic actions that are implementable for a specific grammar with an additional stack for semantic attributes.
- The approach is generic in nature and hence can be extended to other context free grammars.

Stack contents before reduce by production $E \rightarrow E_1 * E_2$

Input string $a := b + c * d$

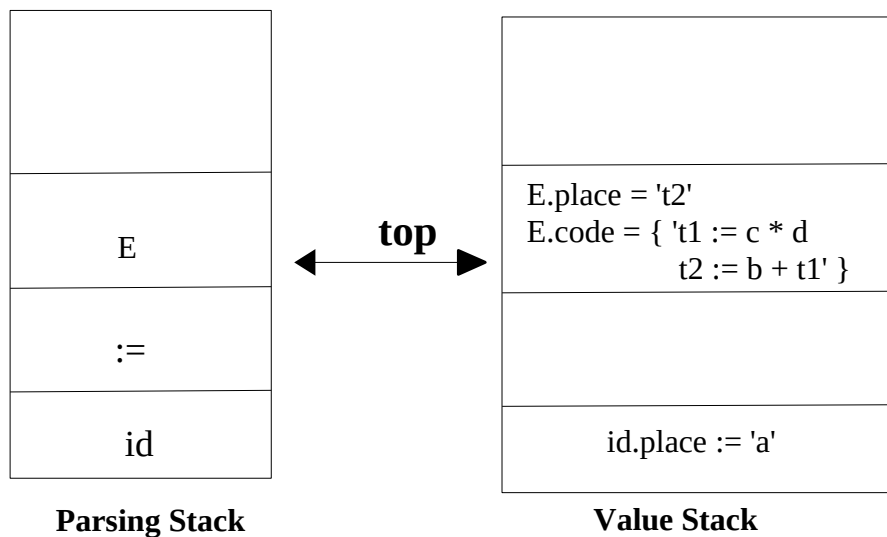


Stack contents immediately after reduction by $E \rightarrow E1 * E2$

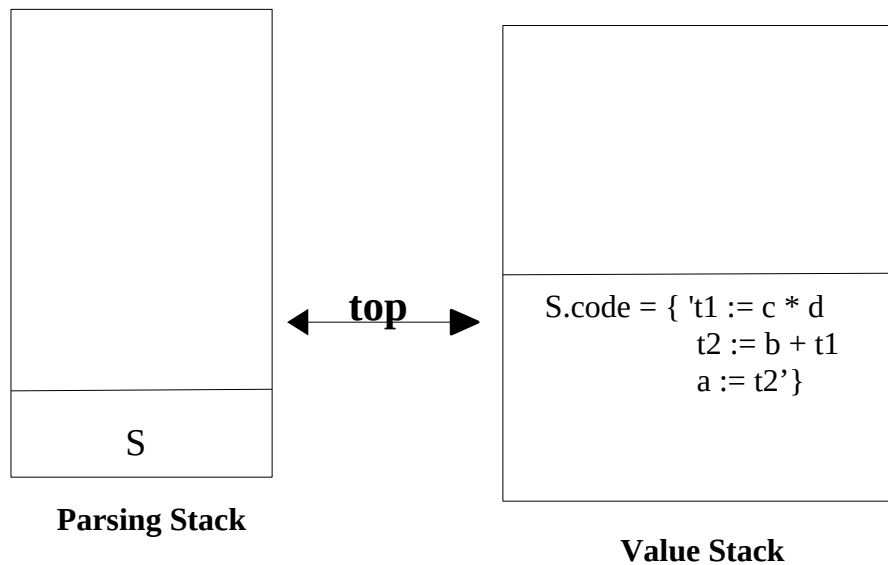


The actions specified in the S-attributed scheme are used to obtain the new stack configuration shown above.

Stack contents immediately after reduction by the production rule $E \rightarrow E1 + E2$



Configuration of the two stacks just after the reduce by the rule $S \rightarrow id := E$, and just before the accept action by the parser is shown below.



METHODS FOR SEMANTIC ANALYSIS

We summarize the different methods introduced in this module for performing semantic analysis in a syntax directed manner.

Carefully distinguish between

- 1) *Syntax directed definition* : A context free grammar augmented with semantic rules. No assumptions are made regarding when the semantic rules are to be evaluated.
- 2) *Translation scheme* : A syntax directed definition with actions instead of semantic rules. Actions are embedded within the right hand side of productions, their positions indicating the time of their evaluation during a parse.
- 3) *Implementation of a syntax directed definition* : A translation scheme in which the actions are described in terms of actual data structures (e.g. value stack) instead of attribute symbols.

Semantic analysis of most programming features covered in this module are demonstrated using translation actions.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

This part of the module deals with writing S-attributed grammars for performing static semantic analysis of various language constructs.

- 1) **Processing of declarations** : defining required attributes, symbol table organization for handling nested procedures and scope information, translation scheme for sample declaration grammars.
- 1) **Type analysis and type checking** : concept of type expression and type equivalence, an algorithm for structural equivalence, performing type checking in expressions and statements and generating intermediate code for expressions.
- 1) **Intermediate code forms** : three address codes and their syntax.
- 2) **Translation of assignment statement.**
- 3) **Translation of Boolean expressions and control flow statements.**

Most of the semantic analysis and translation will be illustrated using synthesized attributes only.

PROCESSING OF DECLARATIONS

Declarations are typically part of the syntactic unit of a procedure or a block. They provide information such as names that are local to the unit, their type, etc.

Semantic analysis and translation involves,

- collecting the names in a symbol table
- entering their attributes such as type, storagerequirement and related information
- checking for uniqueness, etc., as specified by the underlying language
- computing relative addresses for local names which is required for laying out data in the activation record

We begin with a simple grammar for declarations

1. Type associated with a Single Identifier

In the grammar given below, type is specified after an identifier.

- We use a synthesized attribute, **T.type** for storing the type of T.
- **T.width** is another synthesized attribute that denotes the number of memory units taken by an object of this type.
- **id.name** is a synthesized attribute for representing the lexeme associated with id.
- **offset** is used to compute the relative address of an object in the data area. It is a global variable.

PROCESSING OF DECLARATIONS

- Procedure **enter(name, type, width, offset)** creates a symbol table entry for id.name and sets the type, width and offset fields of this entry.

$P \rightarrow MD$
 $M \rightarrow \epsilon \quad \{ \text{offset} := 0 \}$
 $D \rightarrow D; D$
 $D \rightarrow \text{id} : T \quad \{ \text{enter}(\text{id.name}, T.\text{type}, T.\text{width}, \text{offset}); \text{offset} := \text{offset} + T.\text{width} \}$
 $T \rightarrow \text{integer} \quad \{ T.\text{type} := \text{integer}; T.\text{width} := 4 \}$
 $T \rightarrow \text{real} \quad \{ T.\text{type} := \text{real}; T.\text{width} := 8 \}$
 $T \rightarrow \text{id} [\text{num}] \text{ of } T_1$
 $\quad \{ T.\text{width} := \text{num.val} * T_1.\text{width} \}$
 $T \rightarrow \uparrow T_1$
 $\quad \{ T.\text{type} := \text{pointer}(T_1.\text{type}); T.\text{width} := 4 \}$

In the code for semantic analysis given above,

- Arrays are assumed to start at 1 and num.val is a synthesized attribute that gives the upper bound (integer represented by token num).
- If a nonterminal occurs more than once in a rule, its references in the rhs are subscripted and then used in the semantic rules , e.g., use of T in the array declaration.

2. Type associated with a List of Identifiers

If a list of ids is permitted in place of a single id in the above grammar, then

- all the ids must be available when the type information is seen subsequently.
- A possible approach is to maintain a list of such ids, and carry forward a pointer to the list as a synthesized attribute, say, L.list
- The procedure **enter()** given below has a different semantics now which creates a symbol table entry for each member of a list and also sets the type information.
- The two functions, **makelist()** and **append()** perform the tasks of creating a list with one element and for inserting an element in the list respectively. These functions return a pointer to the created / updated list.

$P \rightarrow D$
 $D \rightarrow D; D$
 $D \rightarrow L : T \quad \{ \text{enter}(L.\text{list}, T.\text{type}, T.\text{width}, \text{offset}) \}$
 $L \rightarrow \text{id}, L_1 \quad \{ L.\text{list} := \text{append}(L_1.\text{list}, \text{id.name}) \}$
 $L \rightarrow \text{id} \quad \{ L.\text{list} = \text{makelist}(\text{id.name}) \}$
 $T \rightarrow \text{integer} \quad \{ T.\text{type} := \text{integer}; T.\text{width} := 4 \}$

$$\begin{aligned}
T &\rightarrow \text{real} && \{ T.\text{type} := \text{real}; T.\text{width} := 8 \} \\
T &\rightarrow \text{id} [\text{num}] \text{ of } T_1 && \{ T.\text{width} := \text{num.val} * T_1.\text{width}; \\
&&& T.\text{type} := \text{array} (\text{num.val}, T_1.\text{type}) \} \\
T &\rightarrow \uparrow T_1 && \{ T.\text{type} := \text{pointer} (T_1.\text{type}); T.\text{width} := 4 \}
\end{aligned}$$

3. Type Specified at the beginning of a list

The following grammar provides an example.

$$\begin{aligned}
D &\rightarrow D; D \mid T L \\
L &\rightarrow \text{id}, L \mid \text{id} \\
T &\rightarrow \text{integer} \mid \text{real} \mid \dots
\end{aligned}$$

- The grammar is rewritten to make writing of semantics easier.
- Note the role played by D.syn in propagating the type information

$$\begin{aligned}
D &\rightarrow D; D \\
D &\rightarrow D_1, \text{id} && \{ \text{enter} (\text{id.name}, D_1.\text{syn}); \\
&&& D.\text{syn} := D_1.\text{syn} \} \\
D &\rightarrow T \text{id} && \{ \text{enter} (\text{id.name}, T.\text{type}); \\
&&& D.\text{syn} := T.\text{type} \} \\
T &\rightarrow \text{integer} \mid \text{real} \mid \dots \{ \text{same as earlier} \}
\end{aligned}$$

SCOPE INFORMATION

The method described above can be used to process declarations of names local to a procedure.

For a language that permits nested procedures with lexical scoping, the static scoping rules are as follows. We shall not discuss static scoping rules as they are not applicable in many PLs such as C/C++

- We use the earlier grammar for illustration but add another rule to it.

$$\begin{aligned}
P &\rightarrow D \\
D &\rightarrow D ; D \\
D &\rightarrow \text{id} : T \\
D &\rightarrow \text{proc id} ; D ; S \\
T &\rightarrow \text{integer} \\
T &\rightarrow \text{real} \\
T &\rightarrow \text{array} [\text{num}] \text{ of } T_1 \\
T &\rightarrow \uparrow T_1
\end{aligned}$$

- The rule, $D \rightarrow \text{proc id ; } D ; S$ permits nested procedures, but without parameters.
- The design solution is to create a separate table for handling names local to each procedure. The tables are linked in such a manner that lexical scoping rules are honoured
- The following attributes are defined

id.num T.type T.width

The global variable nest is used to compute the nesting level for a procedure.

- Two new nonterminals, M and N, are introduced so that semantic actions can be written at these points

$P \rightarrow M D$

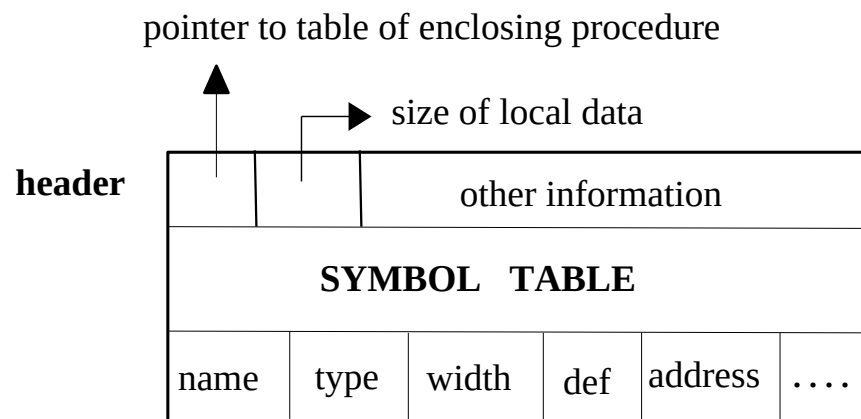
$D \rightarrow \text{proc id ; } N D ; S$

$M \rightarrow \epsilon$

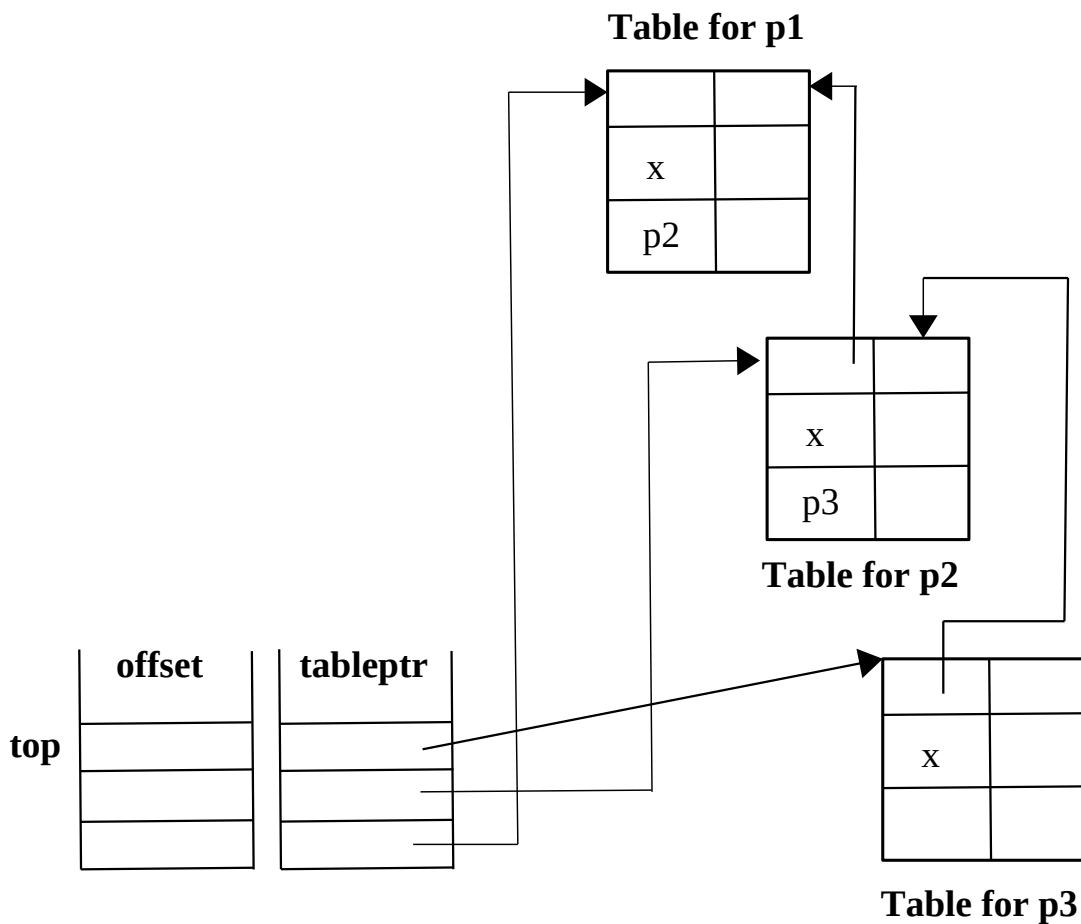
$N \rightarrow \epsilon$

- The place marked by M is used to initialize all information related to the symbol table associated with this level. Similarly N would signal the start of activity for another table.
- The various symbol tables have to be linked properly so as to correctly resolve nonlocal references.
- A stack of symbol tables is used for this purpose.
- The symbol table structures and their interconnection are shown in Figure.
- For symbol table management, the following organization and routines are assumed.
 1. Each table is an array of a suitable size which is linked using the fields as shown.
 2. There are 2 stacks, called **tableptr** and **offset**. The stack tableptr points to the currently active procedure's table and the pointers to the tables of the enclosing procedures are kept below in the stack.
 3. The other stack is used to compute the relative addresses for locals within a procedure.
 4. **mktable(previous)** is a procedure that creates a new table and returns a pointer to it. It also gives the pointer to the previous table through the argument, **previous**. This is required in order to link the new table with that of the closest enclosing procedure's.

Symbol Table Organization for Nested Structures



(b) Organization of a single table



(c) Symbol Tables and their interconnection when control reaches the marked point

width, offset) is a procedure for creating a new entry for the local variable, name, along with its attributes.

5. **enter**(table, name, type, width, offset) is a procedure for creating a new entry for the local variable, name, along with its attributes.
6. **addwidth**(table,width) is used to compute the total space requirement for all the names of a procedure.
7. **enterproc** (table, name, nest level, newtable) creates a new entry for a procedure name. The last argument is a pointer to the corresponding table.

The semantic rules are given now.

$P \rightarrow M D$ {addwidth(top(tableptr),top(offset)) ;
pop(tableptr) ; pop(offset)}

$$\begin{aligned}
M &\rightarrow \varepsilon \{ p := \text{mktable}(\text{nil}); \text{push}(p, \text{tableptr}); \\
&\quad \text{push}(0, \text{offset}); \text{nest} := 1 \} \\
D &\rightarrow D ; D \\
D &\rightarrow \text{id} : T \\
&\quad \{ \text{enter}(\text{top}(\text{tableptr}), \text{id.name}, T.\text{type}, T.\text{width}); \\
&\quad \text{top}(\text{offset}) := \text{top}(\text{offset}) + T.\text{width} \} \\
D &\rightarrow \mathbf{proc} \text{ id} ; N D ; S \\
&\quad \{ p := \text{top}(\text{tableptr}); \text{addwidth}(p, \text{top}(\text{offset})); \\
&\quad \text{pop}(\text{tableptr}) ; \text{pop}(\text{offset}); \\
&\quad \text{enterproc}(\text{top}(\text{tableptr}), \text{id.name}, \text{nest}, p); \\
&\quad \text{nest}--; \\
&\quad \} \\
N &\rightarrow \varepsilon \{ p := \text{mktable}(\text{top}(\text{tableptr})) ; \\
&\quad \text{push}(p, \text{tableptr}); \text{push}(0, \text{offset}) ; \text{nest}++; \\
&\quad \} \\
T &\rightarrow \text{integer} \quad \{ T.\text{type} := \text{integer}; T.\text{width} := 4 \} \\
T &\rightarrow \text{real} \quad \{ T.\text{type} := \text{real}; T.\text{width} := 8 \} \\
T &\rightarrow \text{id} [\text{num}] \text{ of } T_1 \\
&\quad \{ T.\text{width} := \text{num.val} * T_1.\text{width}; \\
&\quad \quad T.\text{type} := \text{array}(\text{num.val}, T_1.\text{type}) \\
&\quad \} \\
T &\rightarrow \uparrow T_1 \\
&\quad \{ T.\text{type} := \text{pointer}(T_1.\text{type}); T.\text{width} := 4 \}
\end{aligned}$$

- The translation scheme described is inadequate to handle recursive procedures. The reason being that the name of such a procedure would be entered in its closest enclosing symbol table only after the entire procedure is parsed.
- However a recursive procedure would have a call to itself within its body and this call cannot be translated since the procedure name would not be present in the table.
- A possible remedy is to enter a procedure's name immediately after it has been found. Semantic actions for only the changed rules are shown.

$$\begin{aligned}
D &\rightarrow \mathbf{proc} \text{ id} ; N D ; S \\
&\quad \{ p := \text{top}(\text{tableptr}); \text{addwidth}(p, \text{top}(\text{offset})); \\
&\quad \text{pop}(\text{tableptr}) ; \text{pop}(\text{offset}); \text{nest}--;
\end{aligned}$$

```

    }
N → ε { p := mktable(top(tableptr)) ;
        enterproc(top(tableptr),id.name, nest, p);
        push(p,tableptr); push(0,offset) ; nest++;
    }

```

SYMBOL TABLE ORGANIZATION

We now consider the issues related to the design and implementation of symbol tables

- Two major factors are the features supported by the programming language and the need to make efficient access to the table.
- The scoping rules of the language indicate whether single or multiple tables are convenient.
- The various types and attributes of names decide the internal organization for an entry in the table.
- Need for efficient access of the table influences the implementation considerations, such as whether arrays, linked lists, binary trees, hash tables or an appropriate mix should be used.
- We examine in brief two possible hash table based implementations in this course.

MULTIPLE HASH TABLES

1. Each symbol table is maintained as a separate hash table.
2. For nonlocal names, several tables may need to be searched which, in turn, may result in searching several chains in case of colliding entries in a hash table.
3. An important issue in this design is the access time for globals
4. Another concern relates to space. The size of each table, if fixed at compile time, could turn out to be a disadvantage, specially if the estimates are wrong.

SINGLE HASH TABLE

A single hash table for block structured languages is possible, provided one stores the scope number associated with each name. A feasible organization is shown in Figure.

- New names are entered at the front of the chains, hence search for a name terminates with the first occurrence on the chain.
- When a scope is closed, the table has to be updated by deleting all entries with the current scope number. The operation is not too expensive, since the search stops at the first entry where the scope values mismatch.
- Basic table operations, enter and search, are efficient because there is a single table. However, storage gains may be compensated due to the inclusion of scope value with each entry.

- All the locals of a scope are not grouped together, hence additional effort in time and space is needed to maintain such information, if so required.

Single hash table organization is depicted in the figure. The entry for a name in a node also contains its scope value.

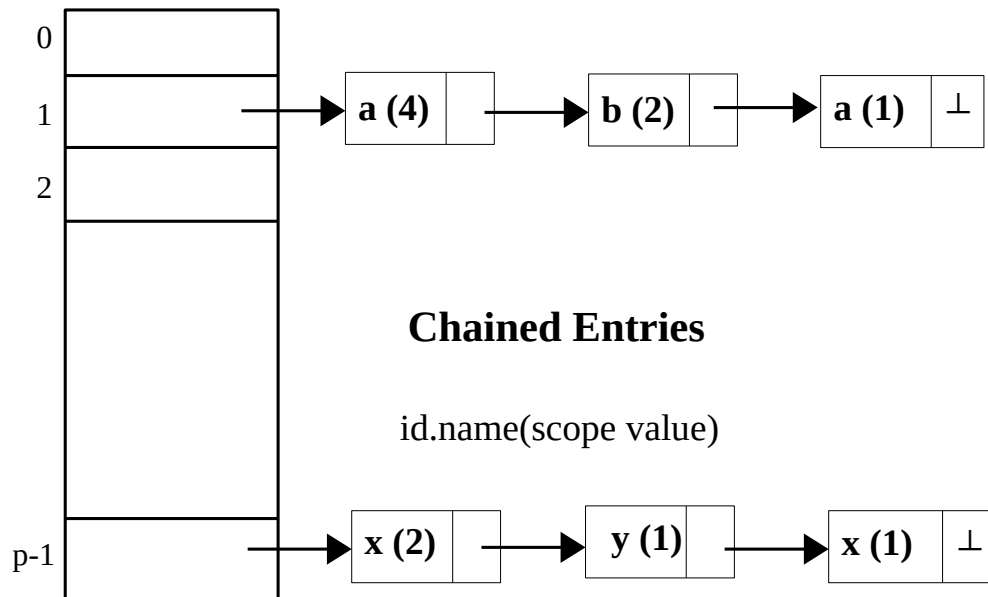


FIGURE :

Single Table for Handling Nested Scopes

The attributes for each name have to be predecided, depending on the processing they undergo in various phases of the compiler.

From the viewpoint of semantic analysis, relevant attributes have been identified in the preceding examples.

- The other phases, like error analysis and handling, code generation and optimization, may introduce more attributes for an entry in the table.

The design discussed for handling of nested procedure declarations can also be used for doing semantic analysis of record declarations. In language C, struct is similar to a record of Pascal.

The earlier grammar is first augmented as given below with a marker nonterminal L. We introduce marker nonterminals to facilitate writing of semantic actions at points of interest.

$$\begin{aligned}
D &\rightarrow D ; D \mid \text{id} : T \\
T &\rightarrow \mathbf{integer} \mid \mathbf{real} \\
T &\rightarrow \mathbf{id} \mid \mathbf{num} \mid \mathbf{of} T1 \\
T &\rightarrow \mathbf{record} L D \mathbf{end} \\
L &\rightarrow \epsilon
\end{aligned}$$

Two stacks tableptr and offset are used in the same sense as introduced earlier.

Attribute T.width gives the width of all the data objects declared within a record.

Attribute T.type is used to hold the type of a record. The expression used here for T.type is explained later during Type Analysis.

The semantic actions for the grammar rules for record are as follows. The actions for other rules remain unchanged.

$$\begin{aligned}
T &\rightarrow \mathbf{record} L D \mathbf{end} \\
&\quad \{ T.type := \text{record}(\text{top}(\text{tableptr}); \\
&\quad \quad T.width := \text{top}(\text{offset}); \\
&\quad \quad \text{pop}(\text{tableptr}); \text{pop}(\text{offset}) \\
&\quad \} \\
L &\rightarrow \epsilon \\
&\quad \{ p := \text{mktable}(\text{nil}); \\
&\quad \quad \text{push}(p, \text{tableptr}); \text{push}(0, \text{offset}) \\
&\quad \}
\end{aligned}$$

Semantic errors that are detected during processing of declarations of a program include the following among many others

- uniqueness checks
- names referenced but not declared
- names declared but not referenced
- names declared but not initialized properly
- ambiguity in declarations
- and many more

Code for detection and handling of such errors are incorporated in the semantics actions at the relevant places.

TYPE ANALYSIS AND TYPE CHECKING

Static type checking is an important part of semantic analysis.

- detect errors arising out of application of an operator to an incompatible operand, and
- to generate intermediate code for expressions and statements.

The notion of types and the rules for assigning types to language constructs are defined by the source language.

Typically, an expression has a type associated with it and types usually have structure.

Usually languages support types of two kinds, basic and constructed.

- Examples of basic types are integer, real, character, boolean and enumerated
- array, record, set and pointer are examples of constructed types. The constructed types have structure.

How to express type of a language construct ?

For basic types, it is straightforward but for the other types it is nontrivial. A convenient form is known as a **type expression**.

1. A **type expression** is defined as follows.
2. A **basic type** is a type expression. Thus boolean, char, integer, real, etc., are type expressions.
For the purposes of type checking, two more basic types are introduced, type **error** and **void**.
3. A **type name** is a type expression. In the example given below, the type name 'table' is a type expression.
type table = array[1 .. 100] of array[1 .. 100] of integer ;
4. A type constructor applied to type expressions is a type expression. The constructors array, product, record, pointer and function are used to create constructed types as described below.

Array : If T is a type expression, then $\text{array}(I, T)$ is also a type expression; array elements are of type T and index set is of type I (usually a range of integer).

Example : For the array declaration

var ROW : array [1 .. 100] of integer;

the type expression for variable ROW is **array(1..100, integer)**

Product : If T_1 and T_2 are type expressions, then their cartesian product $T_1 \times T_2$ is a type

expression; X is left associative. It is used in other constructed types.

Records : The constructor record applied to a product of the type of the fields of a record is a type expression. For example,

```
type entry = record
    token : array [ 1.. 32 ] of char;
    salary : real
end;
var employee : array[1 .. 20] of entry;
```

The type name **entry** has the type expression as given below.

```
record((token X array(1..32, char)) X (salary X real))
```

Exercise : Write the type expression for employee.

Pointers : If T is a type expression, then `pointer(T)` is a type expression. For example for the declaration

```
var ptr : ↑ entry
```

is added to the type name entry, then

type expression for **ptr** is :

```
pointer (record ((token X array(1..32, har)) X (salary X real)))
```

Function : A function in a programming language can be expressed by a mapping $D \rightarrow R$, where D and R are domain and range type.

The type of a function is given by the type expression $D \rightarrow R$. For the declaration :

```
function f (a :real,b:integer) : ↑integer;
```

the type expression for f is

```
real X integer → pointer (integer)
```

Programming languages usually restrict the type a function may return, e.g., range R is not allowed to have arrays and functions in several languages.

How to use the type expression ?

The type expression is a linear form of representation that can conveniently capture structure of a type.

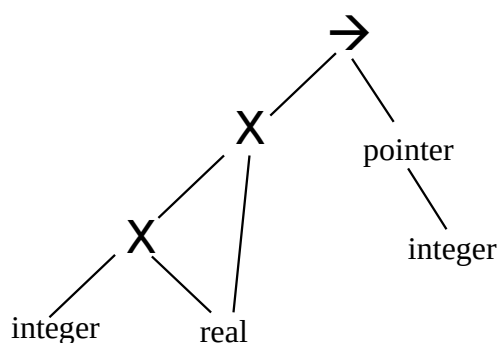
This expression may also be represented in the form of a DAG (Directed Acyclic Graph) or tree.

Consider the function **f (a :integer, b,c : real) : ↑integer;**

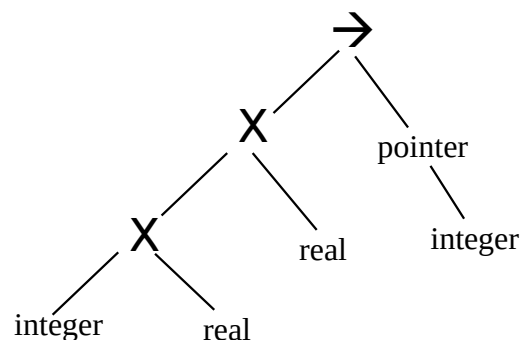
the type expression for f is

integer X real X real → pointer (integer)

Two equivalent representations in the form of a DAG and tree.



DAG form of type graph



Tree form of type graph

Figure : Type Expressions And Type Graphs

The tree (or dag) form, as shown, is more convenient for automatic type checking as compared to the expression form.

A **type system** is a collection of rules for assigning type expressions to linguistic constructs of a program.

A **type checker** implements a type system.

Checking for type compatibility is essentially finding out when two type expressions are equivalent.

For type checking, the general approach is to compare two type expressions and **if two type expressions are equivalent** then return an **appropriate type** else return **type error**.

This, in turn, requires a definition of the notion of **equivalence of type expressions**.

The type expressions of variables in Example 1 are given in the second column.

Example 1 :

```

type link = ↑ node;
var first, last : link;

p : ↑ node;

q , r : ↑ node;

```

Variable	Type	Expression
first	link	
last	link	
p	pointer (node)	
q	pointer (node)	
r	pointer (nod	

Example 2 :

```

type person = record
    id : integer;
    weight : real
end;

type car = record
    id : integer;
    weight : real
end;

var x : person;
var y : car;

```

The issue is to decide whether all the type expressions given above are type equivalent.

Presence of names in type expressions give rise to two distinct notions of equivalence.

One is called **Name Equivalence** and the other **Structural Equivalence**.

Name Equivalence treats each type name as a distinct type, two expressions are name equivalent if and only if they are identical.

For Example 1, the type expressions for variables are reproduced below.

Variable	Type Expression
----------	-----------------

first	link
last	link
p	pointer (node)
q	pointer (node)
r	pointer (node)

Variables **first** and **last** are name equivalent since both have identical type expression, **link**.

Similarly variables p, q and r are name equivalent, but first and p are not.

In Example 2, x and y are not name equivalent because their respective type expressions person are car which are different type names.

If type names are replaced by the type expressions defined by them, then we get the notion of structural equivalence.

Two type expressions are structurally equivalent if they represent structurally equivalent type expressions after the type names have been substituted in.

In Example 1, when the type expression for link is used

Type Name		Type Expression
link		pointer (node)

Variable	Type Expression	Type Name Substitution
first	link	pointer (node)
last	link	pointer (node)
p	pointer (node)	pointer (node)
q	pointer (node)	pointer (node)
r	pointer (node)	pointer (node)

As shown by the third column above, all the variables in Example 1 are structurally equivalent.

In Example 2, the type expressions for type names are

Type Name	Type Expression
person	record ((id X integer) X (weight X real))
car	record ((id X integer) X (weight X real))

After substitution of expression for type names gives

Variable	Type Exp	Type Name Substitution
x	person	record ((id X integer) X (weight X real))
y	car	record ((id X integer) X (weight X real))

Variables x and y are seen to be structurally equivalent.

- Example 2 indicates the problem with structural equivalence. If user declares two different types, they probably represent different objects in the users application. Name equivalent appears to be the right choice here.
- Declaring them type equivalent, merely because they have the same structure, may not be desirable.
- Name equivalence is safer but more restrictive. It is also easy to implement. Compiler only needs to compare the strings representing names of the types.

Two expressions are structurally equivalent if

- i. the expressions are the same basic type, or
- ii. are formed by applying the same constructor to structurally equivalent types

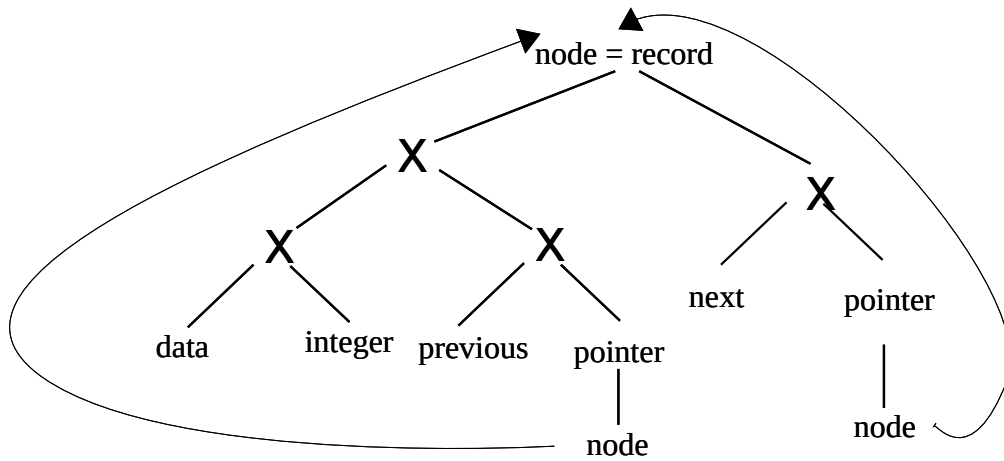
Pascal uses structural equivalence while C uses a mix of both forms of equivalence.

A compiler for a PL that uses structural equivalence would require an algorithm for detecting it at compile time.

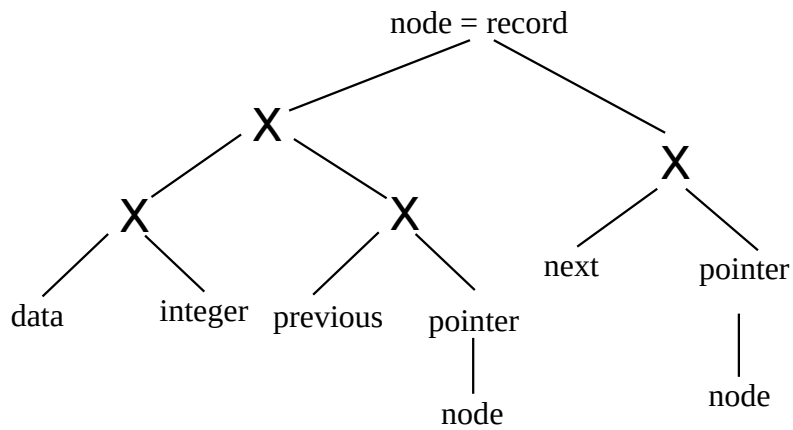
The complexity of such an algorithm is clearly dependent on the structure of the type graphs involved. When the type graph is a tree or a dag, a simpler algorithm is possible. For type graphs containing cycles, such an algorithm is nontrivial.

Cycle in a type graph typically arises due to recursively defined type names. A common situation is depicted by pointers to records.

```
link = ↑ node ;  
node = record  
    data : integer ;  
    previous : link;  
    next : link  
end;
```

Cyclic type graph



Equivalent acyclic graph

IMPLEMENTATION OF STRUCTURAL EQUIVALENCE

- Type graph for type name node is shown in the figure; a cyclic graph is a natural way of representing the tuple expression and it contains two cycles.
- The acyclic equivalent does not substitute the recursive references to node.

Algorithm For Structural Equivalence

A function `sequiv()` is presented in the following for checking structural equivalence of two type expressions (or type graphs).

The function uses two formal parameters, `s` and `t`, which are the input type expressions or roots of the associated type graphs (however the graphs must be acyclic).

The algorithm checks the equivalence of `s` and `t` using

- both are same basic type
- both are arrays of compatible types
- both are compatible cartesian products
- both are compatible pointers, and finally
- both are compatible functions

ALGORITHM FOR STRUCTURAL EQUIVALENCE

```
function sequiv(s,t) : boolean;  
begin  
    if s and t are the same basic type then return true  
  
    else if s = array(s1, s2) and t = array(t1, t2 ) then  
        return sequiv(s1, t1 ) and sequiv(s2, t2)  
  
    else if s = s1 X s2 and t = t1 X t2 then  
        return sequiv(s1, t1 ) and sequiv(s2, t2)  
  
    else if s = pointer(s1) and t = pointer(t1) then  
        return sequiv(s1, t1 )  
  
    else if s = s1 → s2 and t = t1 → t2 then  
        return sequiv(s1, t1 ) and sequiv(s2, t2)  
  
    else return false  
  
end
```

Rewriting the implementation of function sequiv, so that it can be used for detecting structural equivalence of general type graphs is an interesting exercise.

Writing A Type Checker

We use the material on types studied so far to implement a type checker for a sample language given below. The purpose is to demonstrate the working of a type checker only (and hence no code is generated).

```
P → D ; S  
D → D ; D | id : T  
T → char | integer | boolean  
    | id [num] of T | ↑T  
S → id := E | if E then S  
    | while E do S | S ; S  
E → literal | num | id | E mod E
```

| E [E] | E binop E | E ↑

The first rule indicates that the type of each identifier must be declared before being referenced.

The first task is to construct type expressions for each name and code which does that is to be given against the relevant rules.

In case type graphs are used, these actions need to be recoded appropriately. A synthesized attribute, `id.entry`, is used to represent the symbol table entry for `id`.

The procedure `enter()` has the same purpose of installing attributes of a name in the symbol table.

`T.type` is a synthesized attribute that gives the type expression associated with type `T`. Attaching these semantic actions yields the following SDTS

$D \rightarrow id : T$	$\{ \text{enter}(\text{id.entry}, T.\text{type}) \}$
$T \rightarrow \text{char}$	$\{ T.\text{type} = \text{char} \}$
$T \rightarrow \text{integer}$	$\{ T.\text{type} = \text{integer} \}$
$T \rightarrow \text{boolean}$	$\{ T.\text{type} = \text{boolean} \}$
$T \rightarrow \text{array } [\text{num}] \text{ of } T$	$\{ T.\text{type} = \text{array } (1..\text{num.val}, T_1.\text{type}) \}$
$T \rightarrow \uparrow T_1$	$\{ T.\text{type} = \text{pointer}(T_1.\text{type}) \}$

TYPE CHECKING FOR EXPRESSIONS

Once the type expression for each declared data item has been constructed, semantic actions for type checking of expressions can be written, as explained below.

`E.type` is a synthesized attribute that holds the type expression assigned to the expression generated by `E`.

The function `lookup(e)` returns the type expression saved in the symbol table for `e`.

The semantic code given below assigns either the correct type to `E` or the special basic type, **`type_error`**.

The type checker can be enhanced to provide details of the semantic error and its location.

$E \rightarrow$	literal	{E.type := char}
$E \rightarrow$	num	{E.type := integer}
$E \rightarrow$	id	{E.type := lookup (id.entry)}
$E \rightarrow$	$E_1 \text{ mod } E_2$	{E.type := if E_1 .type = integer and sequiv (E_1 .type, E_2 .type) then integer else type_error }
$E \rightarrow$	$E_1 \text{ binop } E_2$	{E.type := if sequiv (E_1 .type, E_2 .type) then E_1 .type else type_error }
$E \rightarrow$	$E_1 [E_2]$	{E.type := if E_2 .type = integer and E_1 .type = array (s, t) then t else type_error }
$E \rightarrow$	$E_1 \uparrow$	{E.type := if E_1 .type = pointer (t) then t else type_error }

TYPE CHECKING FOR STATEMENTS

Semantic code for type checking of the statement related rules of the sample grammar are given in the following.

S.type is a synthesized attribute that is assigned either type **void** or **type_error**, depending upon whether there was a type_error detected within it or not.

The semantic action in the last rule propagates type errors in a sequence of statements.

```
S → id := E      { S.type := if sequiv (id.type, E.type)
                  then void else type_error
                  }

S → if E then S1
    { S.type := if E.type = boolean
    then S1.type else type_error
    }

S → while E do S1
    { S.type := if E.type = boolean
    then S1.type else type_error
    }

S → S1; S2      { S.type := if S1.type = void
                  and S2.type = void
                  then void else type_error
                  }
```

Combining all the grammar rules and the semantic actions listed against them gives a type checker for the sample language.

In the type checker given above, it is assumed that for all operators, the types of the operands are compatible. This assumption is too restrictive and not followed in practice.

For example, **a op i**, where a is a real and i is an integer, is a legal expression in most programming languages.

The compiler is supposed to first convert one of the operands to that of the other and then translate the op.

Language definition specifies, for a given op, the kind of conversions that are permitted. Conversion of integer to real is usually common.

Semantic rules for performing type conversion for expressions and statements are included in the discussion on generation of intermediate code for these constructs.

INTERMEDIATE CODE FORM

The front end typically outputs an explicit form of the source program for subsequent analysis by the back end.

The semantic actions incorporated along with the grammar rules are responsible for emitting them.

Among the several existing intermediate code forms, we shall consider one that is known as Three-Address Code.

In three address code, each statement contains 3 addresses, two for the operands and one for the result. The form is similar to assembly code and such statements may have a symbolic label.

The commonly used three address statements are given below.

1. Assignment statements of the form :
 $x := y \text{ op } z$, where op is binary, or
 $x := \text{op } y$, when op is unary
2. Copy statements of the form $x := y$
3. Unconditional jumps, goto L. The three address code with label L is the next instruction where control flows
4. Conditional jumps, such as
 if $x \text{ relop } y$ goto L
 if x goto L

The first instruction applies a relational operator, relop, to x and y and executes statement with label L, if the relation is true. Else the statement following if $x \text{ relop } y$ goto L is executed. The semantics of the other one is similar.

5. For procedure calls, this language provides the following :

param x

call p, n

where n indicates the number of parameters in the call of p .

6. For handling arrays and indexed statements, it supports statements of the form :

$x := y[i]$
 $x[i] := y$

The first is used to assign to x the value in the location that is i units beyond location y, where x, y and i refer to data objects

7. For pointer and address assignments, it has the statements

$x := \&y$
 $x := *y$
 $*x := y$

In the first form, y should be an object (perhaps an expression) that admits a l-value. In the second one, y is a pointer whose r-value is a location. The last one sets r-value of the object pointed to by x to the r-value of y.

A three address code is an abstract form of intermediate code. It can be realised in several ways, one of them is called **Quadruples**.

A quadruple is a record structure with 4 fields, usually denoted by op, arg1, arg2 and result. For example, the quadruples for the expression $a + b * c$ is

arg1	arg2	result	op	Equivalent form	
*	b	c	t_1		$t_1 = b * c$
+	a	t_1	t_2		$t_2 = a + t_1$

where t_1 and t_2 are compiler generated temporaries.

TRANSLATION OF ASSIGNMENT STATEMENTS

The semantic analysis and translation of assignment statements involve the following activities.

- generating intermediate code for expressions (which are assumed to be free from type_error) and the statement it is a part of,
- generation of temporary names for holding the values of subexpressions during translation,
- handling of array references; performing address calculations for array elements and associating them with the grammar rules,
- performing type conversion (or coercion) operations while translating mixed mode expressions as specified by the underlying language.

We start with a simple grammar and incrementally add features to it. The first grammar, given

below, excludes boolean expressions and array references.

$$P \rightarrow M D$$
$$M \rightarrow \epsilon$$
$$D \rightarrow D ; D \mid \text{id} : T \mid \text{proc id} ; N D ; S$$
$$N \rightarrow \epsilon$$
$$T \rightarrow \text{integer} \mid \text{real}$$
$$S \rightarrow \text{id} := E$$
$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \text{id}$$

The rules for P , D and T give the declaration context in which an assignment statement is to be translated. Expressions of type real and integer only are considered here but other types can be handled similarly.

Ambiguous grammar for expressions have been chosen intentionally, in order to work with a small grammar. The semantic rules can be easily extended to an unambiguous version.

Three address code is generated, wherever required, during semantic analysis. The basic scheme for evaluation of expressions is described in the following.

For an expression of the form $E_1 \text{ op } E_2$

- generate code to evaluate E_1 into a temporary t_1
- generate code to evaluate E_2 into a temporary t_2
- emit code, $t_3 := t_1 \text{ op } t_2$

For compiler generated temporaries, we assume that a new temporary is created whenever required. This is achieved by means of a function `newtemp()` which returns a distinct name on successive invocations.

It is also assumed that temporaries are stored in the symbol table like any other user defined name.

The following synthesized attributes and routines are defined for writing Syntax Directed Translation Scheme (SDTS).

The attribute **id.name** holds the lexeme for the name `id`. The attribute `E.place` holds the value of E .

Function `lookup(id.name)` returns the entry for **id.name** if it exists in the symbol table, otherwise it returns `nil`.

Procedure `emit(statement)` appends the 3-address code **statement** to an output file.

TRANSLATION OF ASSIGNMENT STATEMENTS

$S \rightarrow id := E$	<pre> {p := lookup (id.name); if p ≠ nil then emit (p ':= ' E.place) else error /* undeclared id */ }</pre>
$E \rightarrow E_1 + E_2$	<pre> {E.place := newtemp(); emit (E.place ':= ' E₁.place '+' E₂.place) }</pre>
$E \rightarrow E_1 * E_2$	<pre> {E.place := newtemp(); emit (E.place ':= ' E₁.place '*' E₂.place) }</pre>
$E \rightarrow - E_1$	<pre> {E.place := newtemp(); emit (E.place ':= ' 'uminus' E₁.place) }</pre>
$E \rightarrow (E_1)$	{ E.place := E ₁ .place }
$E \rightarrow id$	<pre> { p := lookup (id.name); if p ≠ nil then E.place := p else error /* undeclared id */ }</pre>

Example : The 3-address code generated for the expression

$$d := - (a + b) * c$$

is given below.

We assume that all variables are of same type and bottom up parsing is used.

Rule used	Attribute Evaluation	Output File
$E \rightarrow id$	E.place := a	-----
$E \rightarrow id$	E.place := b	-----
$E \rightarrow E_1 + E_2$	E.place := t ₁	t ₁ := a + b
$E \rightarrow - E_1$	E.place := t ₂	t ₂ := uminus t ₁
$E \rightarrow E_1 * E_2$	E.place := t ₃	t ₃ := t ₂ * c
$S \rightarrow id := E$	-----	d := t ₃

Type Conversion : In the translation given above, it is assumed that all the operands are of the same type. However that is not always the case.

Languages usually support many types of variables and constants and permit certain operations on operands of mixed types.

Most languages allow an expression of the kind **a op b**, where a or b may be integer or real and op is an arithmetic operation, such as + and *.

For mixed mode expressions, a language specifies whether the operation is legal or not. If illegal, then the compiler has to indicate semantic error.

For legal mixed mode operations, the compiler has to perform type conversion (or type coercion) and then generate appropriate code.

Coercion is an implicit context dependent type conversion performed by a compiler.

To illustrate this concept, we use the earlier grammar where only real and integer types are permitted and convert integers to reals, wherever necessary.

In addition to the attributes mentioned earlier, E.place, id.name, we use E.type that holds the type expression of E.

We assume the existence of an unary operator, **inttoreal**, to perform the desired type conversion from integer value to a real value.

The reverse conversion is not permitted in our framework and hence the function **realtoint** is not needed here.

The function **newtemp()** is now replaced by two functions, **newtemp_int()** and **newtemp_real()** which generate distinct temporaries of type integer and real respectively.

The semantic actions for the rule $E \rightarrow E + E$ is given in the following. The semantics for the other rule $E \rightarrow E * E$ can be written analogously.

The operator + has been further qualified as **real+** or **integer+** depending on the type of operands it acts on.

The semantic actions for the rule $S \rightarrow id := E$

uses a function called as **type_of(p)** whose purpose is to access the symbol table for the name p and return its type.

Structural equivalence is assumed for type checking, whereby **sequiv()** is used.

```

E → E1 + E2
{ if E1.type = integer and sequiv( E1.type, E2.type)
  then begin
    E.place := newtemp_int();
    emit(E.place ':=' E1.place 'int +' E2.place);
    E.type := integer
  end
else if E1.type = real and sequiv( E1.type, E2.type)
  then begin
    E.place := newtemp_real();
    emit(E.place ':=' E1.place 'real +' E2.place);
    E.type := real
  end
else if E1.type = real and E2.type = integer
  then begin
    E.place := newtemp_real();
    t := newtemp_real();
    emit(t ':=' 'inttoreal' E2.place);
    emit( E.place ':=' E1.place 'real +' t);
    E.type := real
  end
else if E1.type = integer and E2.type = real
  then begin
    E.place := newtemp_real();
    t := newtemp_real();
    emit(t ':=' 'inttoreal' E1.place);
    emit( E.place ':=' t 'real +' E2.place);
    E.type := real
  end
else E.type := type_error
}

```

The semantic rules for the assignment statement is now given.

```
S → id := E
{ p := lookup (id.name) ;
  if p = nil then error /* undeclared id */
  else if sequiv (type_of(p), E.type)
    then if E.type = integer
      then begin
        emit ( p 'int:=' E.place) ; S.type := void
      end
    else begin
      emit ( p 'real:=' E.place);
      S.type := void
    end
  else if type_of (p) = real and E.type = integer
    then begin
      t := newtemp_real ();
      emit ( t ':=' 'inttoreal' E.place);
      emit ( p 'real:= ' t ) ;
      S.type := void;
    end
  else S.type := type error; /* type mismatch */
}
```

Note that the temporaries used above are of different types.

Only temporary t1 is of type integer and the rest are all real.

Example : A program fragment, alongwith the 3-address code generated for it, according to the SDTS, is given below.

Program Fragment

```
var a, b, : integer ;
var d : real ;
a := b * c + b * d ;
```

Intermediate Code

```
t1 := b int* c
t2 := inttoreal b
t3 := t2 real* d
t4 := inttoreal t1
t5 := t4 real+ t3
type error
```

The type_error is indicated because **a** is of type **integer** while the type of rhs , i.e., **t5** is **real**. This is our interpretation that is used in the translation above (integer variable can not be assigned a real value).

In practice, a language defines the type conversion rules applicable. Therefore a language may coerce the rhs to an integer value and then assign it to the lhs integer name.

Check out what your language and compiler do in this situation ?

Exercise : Change the semantic rules to what your language defines for such assignments.

TRANSLATION OF ARRAY REFERENCES

We now augment the grammar to include array references as well. The basic issues for arrays are explained through an example.

Consider the following declaration and assignment statement.

```
a : array [ 1..10, 1..20] of integer
i, j, k : integer
i := ____; j := ____;
k := a[i+2, j-5] ;
```

In order to translate the array reference given above, we must know the address of this array element at compile time.

For statically declared arrays, it is possible to compute the relative address of each element (Data Structures course).

- Array is usually stored in contiguous locations.
- There are two possible layouts of memory, known as, row-major representation (used in Pascal /C/C++) and column-major representation (used in Fortran).
- In Fortran, array is indexed from 1; C/C++ start with index 0; there are languages that permit negative indices also.

TRANSLATION OF ARRAY REFERENCES

The row-major and column major forms for a 2 dimensional array of size 3x3 are shown below.

1	2	3	4	5	6	7	8	9
A[1, 1]	A[1, 2]	A[1, 3]	A[2, 1]	A[2, 2]	A[2, 3]	A[3, 1]	A[3, 2]	A[3, 3]

1	2	3	4	5	6	7	8	9
A[1, 1]	A[2, 1]	A[3, 1]	A[1, 2]	A[2, 2]	A[3, 2]	A[1, 3]	A[2, 3]	A[3, 3]

Column major representation of 2D array

Address calculation at compile time

Let base be the relative address of a[1,1]. Then the relative address of a[i₁, i₂] is given by

$$\text{base} + ((i_1 - 1) * 20 + i_2 - 1) * w$$

where w is the width of an array element (space for an element in bytes).

The above expression may be rewritten as

$$c + (20 * i_1 + i_2) * w$$

where c = base - 21 * w

TRANSLATION OF ARRAY REFERENCES

The term, $c = \text{base} - 21 * w$, is a compilation time constant and can be pre computed and stored in the symbol table, while processing array declarations.

Assuming that c has been precomputed and saved in the symbol table for the array $a[]$ of our example, the 3-address code generated for the array assignment

$k := a[i + 2, j - 5];$

using address of $a[i_1, i_2]$ as $c + (20 * i_1 + i_2) * w$, is

$t_1 := i + 2$

$t_2 := t_1 * 20$

$t_3 := j - 5$

$t_4 := t_2 + t_3$

$t_5 := 4 * t_4$ {assuming w is 4}

$t_6 := c$

$t_7 := t_6[t_5]$

$k := t_7$

In the following we write a suitable grammar for array references and attach semantic actions to it.

TRANSLATION OF ARRAY REFERENCES

The address computation has two parts, one is a compile time constant and the other is variable part dependent on the index expressions.

Generalization of the address computation formula for n-dimensional arrays is required.

Let the declaration of a n-dimensional array be of the form : $a[u_1, u_2, \dots, u_n]$

the lower bounds for each dimension is assumed to be 1.

In this setting, $a[10, 20, 30, 40]$ declares a 4-dimensional array with 240000 elements which are indexed from $a[1, 1, 1, 1]$ through $a[10, 20, 30, 40]$.

An array reference, $a[e_1, e_2, \dots, e_n]$, is given with e_i , $1 \leq i \leq n$, being the index expression at the i^{th} subscript position. The address of $a[1, 1, \dots, 1]$ is the base address.

Relative address of $a[e_1, e_2, \dots, e_n] = c + v$

where $c = \text{base} - (((\dots ((u_2 + 1) * u_3 + 1) * \dots) * u_n + 1) * w)$, and

$$v = (((\dots ((e_1 * u_2 + e_2) * u_3 + e_3) * \dots) * u_n + e_n) * w)$$

Q1. How does one derive such expressions ?

Q2. How does the compiler use these to calculate addresses?

TRANSLATION OF ARRAY REFERENCES

In order to calculate v incrementally, the following observation is useful.

Assume an array reference $a[e_1, e_2, \dots, e_n]$

Expression seen	Code Generated
$a[e_1$	e_1 is evaluated in t_1
e_2	e_2 is evaluated in t_2
	$t_3 := t_1 * u_2$
\dots	$t_3 := t_3 + t_2$
\dots	\dots
e_n	\dots
	e_n is evaluated in t_{2n-2}
	$t_{2n-1} := t_{2n-3} * u_n$
	$t_{2n-1} := t_{2n-1} + t_{2n-2}, n \geq 2$
	$t_{2n} := c$

At this stage the reference may be replaced by $t_{2n}[t_{2n+1}]$.

Note that this scheme, as given, requires a large number of temporaries.

In view of the translation scheme outlined above, we need to write a grammar that exposes the index expressions one by one.

As an index expression is parsed the semantic rules should generate intermediate code for the address calculation for that expression.

TRANSLATION OF ARRAY REFERENCES

The skeleton grammar is given below.

$$S \rightarrow L := E$$
$$E \rightarrow L \mid \text{other rules for expression}$$
$$L \rightarrow \text{elist }] \mid \text{id}$$
$$\text{elist} \rightarrow \text{elist } , E \mid \text{id } [E$$

The rules for L take care of array references in both the lhs and rhs of an assignment statement.

The rules for the nonterminal elist are written in such a manner so as to match the intended translation scheme.

The semantic actions that need to be taken against the grammar rules are given.

1. The rule **elist** \rightarrow **id** [**E** captures the array name and the first index expression. The location which holds the value of E must be noted in order to compute v.
2. The rule **elist** \rightarrow **elist** , **E** exposes one by one the index expressions encountered from left to right. The part of v that has been computed so far is kept as an attribute of elist.

TRANSLATION OF ARRAY REFERENCES

3. Using the current value of E , in $\mathbf{elist} \rightarrow \mathbf{elist}, E$ the term corresponding to the dimension under consideration can now be added to v .

Since each term of the address computation formula requires the corresponding upper bound, it is necessary to keep track of the dimension that corresponds to this occurrence of E .

4. The rule, $\mathbf{L} \rightarrow \mathbf{elist} \mathbf{J}$ indicates that an array reference has been completely seen. The two parts c and v of the address computation are available at this point and can be propagated as attributes of L .
5. The rule, $\mathbf{E} \rightarrow \mathbf{L}$ indicates that instance of an array reference (r-value) has been found. The appropriate code can be generated and location where this value is held can be saved as an attribute of E .
6. The rule, $\mathbf{S} \rightarrow \mathbf{L} := \mathbf{E}$ indicates an array reference whose lvalue is required. Appropriate code may be generated since all the required information is available.

TRANSLATION OF ARRAY REFERENCES

SDTS for array references are given below.

$S \rightarrow L := E$

```
{ if L.offset = null
  then emit (L.place ':=' E.place)
  else emit (L.place '[' L.offset ']' ':=' E.place)
}
```

$E \rightarrow L$

```
{ if L.offset = null
  then E.place := L.place)
  else begin
      E.place := newtemp();
      emit (E.place ':=' L.place '[' L.offset ']')
  end
}
```

$L \rightarrow id$

```
{ L.place := id.place;
  L.offset := null
}
```

TRANSLATION OF ARRAY REFERENCES

$L \rightarrow \text{elist }]$

```
{ L.place := newtemp();  
  L.offset := newtemp() ;  
  emit(L.place ':=' c(elist.array));  
  emit(L.offset ':=' elist.place '*' width(elist.array))  
}
```

$\text{elist} \rightarrow \text{elist}_1, E$

```
{ t := newtemp() ;  
  m := elist1.dim + 1 ;  
  emit( t ':=' elist1.place '*' limit(elist1.array, m));  
  emit( t ':=' t '+' E.place);  
  elist.array := elist1.array;  
  elist.place := t ; elist.dim := m  
}
```

$\text{elist} \rightarrow \text{id } [E$

```
{ elist.array := id.place;  
  elist.place := E.place; elist.dim := 1  
}
```

TRANSLATION OF ARRAY REFERENCES

Brief justifications about the attributes and the functions used in our SDTS.

For L the attributes, L.place and L.offset hold the values of c and v in case of arrays. In case of scalars, the second attribute has null value.

For E, the attribute E.place is used in the same sense.

For elist, the attributes used are elist.array, elist.place and elist.dim.

- Attribute **elist.array** is a pointer to the symbol table entry for the array name.
- Attribute **elist.place** is a placeholder for the v part of the address calculation performed so far.
- **elist.dim** is used to keep track of the dimension information.
- The function **c(array_name)** returns the c part of the address formula from the symbol table entry for array name.
- The function **width(array_name)** gives the width information from the symbol table.
- The function **limit(array_name, m)** returns the upper bound for the dimension m from the symbol table.

TRANSLATION OF ARRAY REFERENCES

Calculation of c and v parts of address formula

The complex part of c in

$$c = \text{base} - ((\dots ((u_2+1)*u_3+1)*\dots)*u_n+1) * w$$

is the part $((\dots ((u_2+1)*u_3+1)*\dots)*u_n+1)$

The calculations work out like

$$x_1 = 1; x_2 = x_1 * u_2 + 1; x_3 = x_2 * u_3 + 1; \dots$$

which leads to recurrence relation

$$x_1 = 1;$$

$$x_{i+1} = x_i * u_{i+1} + 1; i > 1$$

The above recurrence formulation can be used to incrementally compute c after parsing a index expression at a given dimension d.

$$c = \text{base} - x_d * w$$

The v part of the address calculation is captured by another recurrence relation

$$y_1 = e_1; y_{i+1} = y_i * u_{i+1} + e_i; i > 1$$

$$v = y_i * w; \text{ after } i\text{th index expression is parsed.}$$

Note : The SDTS given above is a generic array reference translation scheme where the arrays start with index 1 and the representation is row-major.

However C / C++ have the following differences.

- Array reference syntax is $a[e_1][e_2] \dots [e_n]$ instead of $a[e_1, e_2, \dots, e_n]$
- Starting index is 0 and not 1

Upper bounds of any dimension is ≥ 0 (the lower bound), similar to that used for the general case.

In the following, we shall write SDTS for array reference in C/C++.

Let the declaration of a n-dimensional array be of the form :

$$a[u_1] [u_2] [\dots [u_n] ; \text{ the lower bounds for each dimension is } \geq 0.$$

Consider the array reference $a[e_1] [e_2] [\dots [e_n]$, where e_i is the subscript expression at the i^{th} index position. The start address of the array is referred to as $\text{base}(a)$, and the compiler is aware of where it will layout array at run time. The symbol w denotes the size in bytes of an element of the array and it depends on the type of the array.

Array reference	Number of dimension	Address of the reference $a[] [] \dots []$
-----------------	---------------------	---

$a[e_1]$	1	$\text{base}(a) + e_1 * w$
$a[e_1][e_2]$	2	$\text{base}(a) + (e_1 * U_2 + e_2) * w$
$a[e_1][e_2][e_3]$	3	$\text{base}(a) + (e_1 * U_2 * U_3 + e_2 * U_3 + e_3) * w$

The formulation in the address expression is captured by the following precise mathematical

$$\text{addr}(a[e_1][e_2] \dots [e_n]) = \text{base}(a) + \left(\sum_{i=1}^{i=n} e_i * \prod_{j=i+1}^{j=n} U_j \right) * w \quad \dots\dots (1)$$

You may observe that putting $n = 1$ gives the address expression for dimension 1 and so on for higher dimensions. The key part of the address calculation is the expression :

$$\sum_{i=1}^{i=n} e_i * \prod_{j=i+1}^{j=n} U_j \quad \dots\dots\dots (2)$$

which is calculation intensive. Fortunately the same expression can be expressed using the recurrence relation given below.

$$y_1 = e_1; y_n = y_{n-1} * U_n + e_n; n \geq 2 \quad \dots (3)$$

There are two distinct advantages of Equation (3) over that of (2).

- The calculation of y_i from y_{i-1} requires 1 addition and 1 multiplication, as a result computation of y_n requires $n-1$ additions and $n-1$ products, a linear function of the operations.
- The address calculation can be constructed in an incremental fashion as the index expressions are encountered one by one.

The incremental scheme is explicated in the table below.

Array reference seen so far	Key part of address calculation	Intermediate code
$a[e_1]$	e_1	$t_1 := e_1$
$a[e_1][e_2]$	$e_1 * U_2 + e_2$	$t_2 := e_2$ $t_1 := t_1 * U_2$ $t_2 := t_1 + t_2$
$a[e_1][e_2][e_3]$	$(e_1 * U_2 + e_2) * U_3 + e_3$	$t_3 := e_3$ $t_2 := t_2 * U_3$ $t_3 := t_2 + t_3$
$a[e_1][e_2][e_3][e_n]$	Let t_{n-1} be the place which holds the evaluation upto e_{n-1} $t_n := e_n$ $t_{n-1} := t_{n-1} * U_n$ $t_n := t_{n-1} + t_n$

$a[e_1] [e_2] [e_3] [e_n]$	$t_{n+1} := \text{base}(a)$ $t_n := t_n * w$ $t_{n+2} := t_{n+1} [t_n]$
----------------------------	-------	---

SDTS requires a grammar, which enables the translation outlined above.

```

S → L := E
E → L          ## array reference in the rhs of assignment
    | E + E | E * E | ... other expressions for E
L → elist ]     ## last subscript expression
    | id        ## scalar not an array reference
elist → elist [ E ## one more subscript expression
        | id [ E  ## first subscript expression

```

Like the generic array reference case, we need the support of the symbol table and related information about the array declaration here. The same support functions are assumed to be available for getting information from the symbol table, such as name of the array, bounds at each dimension, width of an array element, etc. The SDTS is given below.

```

S → L = E { if L.offset = null
            then emit (L.place ':=' E.place)
            else emit (L.place '[' L.offset ']' ':=' E.place)
            }

E → L { if L.offset = null
        then E.place := L.place)
        else
            E.place := newtemp();
            emit (E.place ':=' L.place '[' L.offset ']')
        end
    }

```

```

L → id { L.place := id.place; L.offset := null }

```

```

L → elist ]
    { L.place := newtemp();
      L.offset := newtemp() ;
      emit(L.place ':=' base(elist.array));
    }

```

```

    emit(L.offset ':=' elist.place '*' width(elist.array))
}

```

```

elist → elist1 [ E
    { m := elist1.dim + 1 ;
      emit( elist1.place ':=' elist1.place '*' limit(elist1.array, m));
      emit( E.place ':=' E.place '+' elist1.place );
      elist.array := elist1.array; elist.place := E.place ; elist.dim := m
    }

```

```

elist → id [ E
    { elist.array := id.place;
      elist.place := E.place; elist.dim := 1
    }

```

The SDTS above uses less temporaries than the one given for the generic case.

Exercise : Show the intermediate code generated for the assignment :

$$b[i+2][2*j-1] = x + a[i*i][j+10][k=2];$$

TRANSLATION OF BOOLEAN EXPRESSIONS

We wish to enrich our grammar by permitting boolean expressions in an assignment statement. The following rules

are representative for our purpose.

```

B →  B or B | B and B
    | not B | ( B )
    | true | false
    | E relop E | E

```

$E \rightarrow E + E \mid \text{other rules for arithmetic expressions}$

The rules for B give the syntax of boolean expressions. For arithmetic expressions, the same rules for E are assumed.

The term **relop** stands for the class of relational operators such as $\{ <, <=, ==, !=, >, >= \}$.

The common approach to translate boolean expressions (in the context of an assignment) is to encode **true** and **false** numerically and use the general translation scheme for arithmetic expressions.

We shall demonstrate the translation based on numerical encoding by using **1** for true and **0** for false.

TRANSLATION OF BOOLEAN EXPRESSIONS

Semantic actions for the rule $B \rightarrow E_1 \text{ relop } E_2$ is the only one which needs explaining.

We generate a temporary, say t , for holding the value of the expression $E_1 \text{ relop } E_2$.

The value assigned to t is either 1 or 0 depending on whether the expression is true or false.

In either case, t is carried forward in the translation of the rest of the expression.

We assume that all three address statements are labeled numerically and that `nextstat` gives the label of the next 3 address statement.

By using the unconditional and conditional branching 3 address statements and `nextstat`, the semantic actions for this rule are written.

The rules for $B \rightarrow \text{true} \mid \text{false}$ cause the generation of a new temporary initialized to 1 or 0 respectively.

The rules for $B \rightarrow B \text{ or } B \mid B \text{ and } B$ cause generation of code to evaluate the boolean expressions, **or** and **and** are operators available in our intermediate code language.

TRANSLATION OF BOOLEAN EXPRESSIONS

SDTS for Boolean expressions is now given.

$B \rightarrow B_1 \textbf{ or } B_2$

```
{ B.place := newtemp();  
  emit(B.place ':=' B1.place 'or' B2.place)  
}
```

$B \rightarrow B_1 \textbf{ and } B_2$

```
{ B.place := newtemp();  
  emit(B.place ':=' B1.place 'and' B2.place)  
}
```

$B \rightarrow \textbf{not } B_1$

```
{ B.place := newtemp();  
  emit(B.place ':=' 'not' B1.place )  
}
```

$B \rightarrow (B_1)$

```
{ B.place := B1.place }
```

$B \rightarrow \textbf{true}$

```
{ B.place := newtemp();  
  emit (B.place ':=' '1')  
}
```

$B \rightarrow \textbf{false}$

```
{ B.place := newtemp();  
  emit (B.place ':=' '0')  
}
```

$B \rightarrow E$

```
{ B.place := E.place }
```

$B \rightarrow E_1 \textbf{ relop } E_2$

```
{ B.place := newtemp();  
  emit ('if' E1.place 'relop.op' E2.place 'goto' nextstat+3);  
  emit (B.place ':=' '0');
```

```
emit ('goto' nextstat + 2);  
emit (B.place ':=' '1')  
}
```

The attribute **op** in **relop.op** is used to distinguish the relational operator in its class.

The sequence of emit statements in the semantic actions for the last rule gives the rationale behind the usage of nextstat+2 or nextstat+3.

The method used in the SDTS above evaluates a boolean expression completely is a safe implementation (generated code preserves semantics).

We shall refer to this method as **full** or **complete evaluation** of a boolean expression.

The translation ignores issues of type checking and type coercion. Such rules are required to correctly handle expressions that have mixed mode operands, one operand being boolean and other permitted numeric type in the language.

Semantic rules for type conversion is discussed in the context of arithmetic expressions. Similar rules will have to be written for boolean expressions that have permissible mixed mode operands.

SUMMARY OF SDTS FOR ASSIGNMENT STATEMENT

We have covered a wide range of issues dealing with translation of an assignment statement. However the treatment is not exhaustive and several details have been skipped for sake of clarity and brevity.

To be able to write a general grammar that includes expressions of all permissible types, such as boolean, integer, real and other numeric types.

Though the salient issues of translation have been exhibited with ambiguous grammars, it is desirable that unambiguous grammars that directly take care of properties of operators, such as precedence and others, be used.

The type checking and type coercion part of semantic analysis need to be integrated and extended over all permissible types.

The translation of array references made assumptions about lower bounds, availability of c and type checking/ coercion.

The scheme needs to be elaborated by appropriate generalisations, and inclusion of missing analyses.

Once the design for the basic grammar is complete, other features may be added along the same lines. For instance, record accesses and function calls are common in expressions.

SUMMARY OF SDTS FOR ASSIGNMENT STATEMENT

The approach to incorporate more features comprises of the following activities.

- Add relevant rules to the existing grammar.
- Decide on the 3-address codes to be generated.
- Identify type checking and type coercion, if necessary, and insert semantic rules to that effect.
- Function calls may require elaborate type checking of all arguments and appropriate translation of parameter passing mechanisms employed.
- Integrate the grammar and semantic actions with the existing one.
- Detection of semantic errors at various places have been indicated. More elaborate detection of such errors, issuing of appropriate error messages and possible strategies for recovery need to be worked out and incorporated.

It should be obvious that semantic analysis for expressions is a nontrivial task and comprises the bulk of semantic analysis for the entire language.

SEMANTIC ANALYSIS USING S-ATTRIBUTES

Translation Of Control Flow Constructs

Translation and generation of intermediate code for typical control flow constructs is considered here. We use the following grammar.

S	→	if B then S	
		if B then S else S	
		while B do S	
		begin Slist end	
		L := E	
Slist	→	Slist ; S	
		S	
B	→	B ₁ or B ₂	B ₁ and B ₂
		not B ₁	(B ₁)
		true	false
		E ₁ relop E ₂	E
E	→	E ₁ + E ₂	other rules for expression
L	→	rules for lhs in assignment statement	

TRANSLATION OF CONTROL FLOW CONSTRUCTS

A comment about the grammar. It is capable of generating nested control flow statements such as nested if-then-else, nested while, etc.

The nonterminals used have the following purpose :

S Start nonterminal and also nonterminal for some type of statements – assignment and control flow

Slist List of one or more statements of type S

B Boolean expression

relop terminal symbol for relational operators, such as {<, <=, >, >=, =, !=}

The rules for assignment statement are assumed without elaboration.

Rule for a compound statement enclosed in begin and end is also included.

The translation issues are introduced through an example.

Example : Consider the boolean expression given below.

$a+b > c$ **and** flag

where a,b and c are integer variables and flag is boolean.

Equivalent 3-address code sequence is given below.

```
10 :   t1 := a + b
11 :   if t1 > c goto 14
12 :   t2 := 0
13 : goto 15
14 :   t2 := 1
15 :   t3 := t2 and flag
```

Assume that the expression $a+b > c$ **and** flag is part of an if-then-else statement, of the form below.

```
If  $a+b > c$  and flag then
begin
```

```

        a := b + c;
        flag := false
    end
    else a := b - c ;

```

In order to generate code for above statement, it is required to append the following two statements after the code for expression evaluation.

The two labels enable the control to flow to the then and else parts respectively.

```
16 :   if t3 goto label1
```

```
17 :   goto label2
```

The statements that need to be generated for the then and else parts are given below. Note the inclusion of a jump statement after the then part, in order to prevent control flow to the else part.

Code for Then part

```
18 : t4 := b + c
```

```
19 : a := t4
```

```
20 : t5 := 0
```

```
21 : flag := t5
```

```
22 : goto label3
```

Code for Else part

```
23 : t6 := b - c
```

```
24 : a := t6
```

```
25 :
```

It is clear that label1 should be 18 (or nextstat + 2) and may be filled in during generation of the statement 16.

TRANSLATION OF CONTROL FLOW CONSTRUCTS

The symbol label2 enables a jump to the beginning of the else part, i.e., 23 in our example. However when the statement 17 is being generated, it is not possible to know the value of this label.

The value clearly depends upon the number of statements in the then part.

Similarly, the value of label3 should be 25, where the code for the statement immediately following the if-then-else, would be placed.

Again its value depends on the number of statements in the else part and can not be resolved while generating statement 22.

The issues for translation of the if-then-else statement may be summarized to

- decide the code sequences for 3 components of this construct during parsing - the conditional, statements in the then-part, and statements in else part. Finally the inclusion of an unconditional jump after the then part to ensure control does not fall through into the else part.
- find a scheme to resolve the forward labels of the jump statements, such as label2 and label3, since the relevant information is not available till all the 3 components have been seen.

Instead of using the method of complete evaluation of a boolean expression, it is possible to use another method which is known as partial evaluation.

In partial evaluation of boolean expressions, the value of the expression is not explicitly stored and it may not even be evaluated completely.

The essence of partial evaluation can be explained by considering the two 3-address code sequences given below.

Code fragment :

```
If a+b > c and flag then  
  begin  
    a := b + c;  
    flag := false
```

end

else $a := b - c$;

Complete Evaluation

```
10 :   t1 := a + b
11 :   if t1 > c goto 14
12 :   t2 := 0
13 :   goto 15
14 :   t2 := 1
15 :   t3 := t2 and flag
16 :   if t3 goto 18
17 :   goto 23
18 :   t4 := b + c
19 :   a := t4
20 :   t5 := 0
21 :   flag := t5
22 :   goto 25
23 :   t6 := b - c
24 : a := t6
25 :
```

Partial Evaluation

```
50 :   t1 := a + b
51 :   if t1 > c goto 53
52 :   goto 60
53 :   if flag goto 55
54 :   goto 60
55 :   t2 := b + c
56 :   a := t2
57 :   t3 := 0
58 :   flag := t3
59 :   goto 62
60 :   t4 := b - c
61 :   a := t4
62 :
```

The code generated for partial evaluation does not use the temporaries t2 and t3 to hold the values of subexpressions as they have been used in the first column.

Using the fact that for an expression of the form $b1 \text{ and } b2$, evaluate $b2$ only when $b1$ is true. Similarly for an expression of the form $b1 \text{ or } b2$, evaluate $b2$ only when $b1$ is false.

The code in the second column shows that it needs less number of temporaries and is also shorter in length as compared to that in the first column.

There are more jump statements in the code for partial evaluation. The number of forward resolutions of jump labels are also correspondingly higher in the second column.

It may be noted that in case of complete evaluation the generated code for expression has exactly two unresolved jump labels.

In order to translate using this method, the issues are

- i. to select code sequences to be generated for the subexpressions of the form **e1 relop e2 ; b1 and b2**, etc., and
- ii. To resolve the target labels of jump statements. The context in which the expression occurs , e.g., **if-then-else**, **while**, also play a role in this part.

It may be noted that target labels of some jump statements may be resolved at generation time itself.

```
50 :   t1 := a + b
51 :   if t1 > c goto 53
52 :   goto 60
53 :   if flag goto 55
54 :   goto 60
```

Statement 51, shown above, is an example. The fact is that jump statements whose targets lie locally (within the code for the expression) can be resolved easily.

Jump statements whose targets are beyond the expression, such as the beginning of then or else part, can not be resolved in this manner.

For example, after processing the expression $a+b > c$ and flag, the statements numbered 50 to 54 are generated.

However the target labels of statements 52 and 54 are not known at this stage.

Also the target of 53 is known at generation time only because of the context of the if-then-else statement, and may not be known in a different situation.

The unresolved labels occurring in statements 52 and 54 have a common characteristic , the expression evaluates to false at these points in the code.

A general method to resolve such labels would be to remember all the statement numbers where the expression evaluates to false. When the target becomes known later it can be inserted in all these statements.

Why did we not need to remember all the statement numbers where the expression evaluates to true ?

This problem did not surface in our example. However, it can be easily seen that in general, we need to keep this information as well (try the same example, replacing **and** by **or**).

The observations above determine the actions that are required for partial evaluation of an boolean expression and to attach these actions at the appropriate place during the translation of control flow constructs.

- A CFG that generates boolean expressions in the context of their use in control flow

structures is written first.

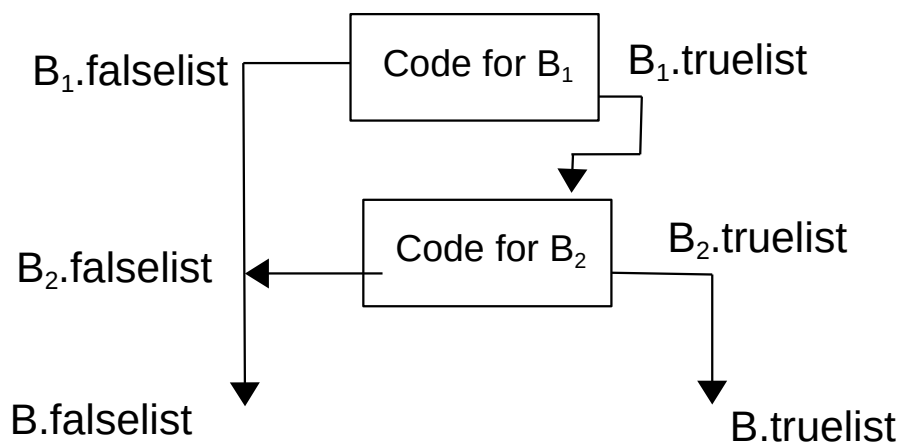
- Semantic actions are then added to generate code using partial evaluation of a boolean expression

SEMANTIC RULES FOR BOOLEAN EXPRESSIONS

A grammar alongwith semantic rules for partial evaluation of Boolean expressions is given below.

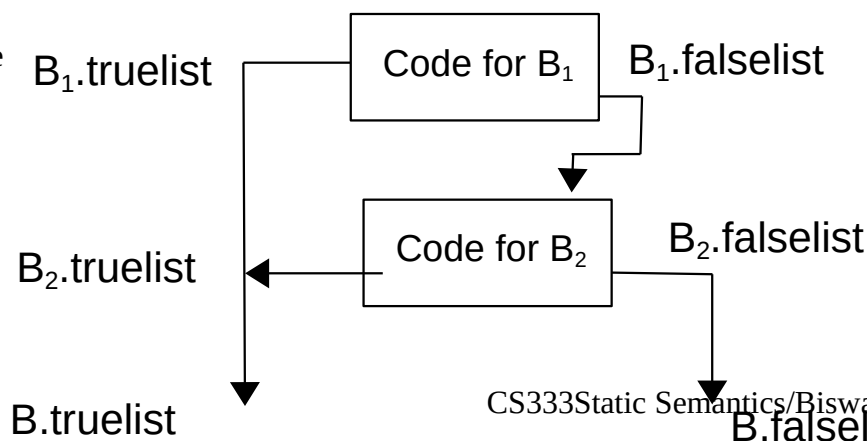
$B \rightarrow B_1 \text{ or } B_2$
 $B \rightarrow B_1 \text{ and } B_2$
 $B \rightarrow \text{not } B_1$
 $B \rightarrow (B_1)$
 $B \rightarrow \text{true} \mid \text{false}$
 $B \rightarrow E_1 \text{ relop } E_2 \mid E$
 $E \rightarrow E_1 + E_2 \mid \text{other rules}$

Consider the second rule, $B \rightarrow B_1 \text{ and } B_2$. Assume that parsing and semantic actions for the right hand side of this rule is complete and available. The scheme for linking the code sequences of the rhs in order to construct that of the lhs nonterminal B is illustrated in the figure below.



Actions for $B \rightarrow B_1 \text{ and } B_2$

Similar linkage rule is as below.



Actions for $B \rightarrow B_1 \text{ or } B_2$

The purpose of the semantic attributes used above are explained here.

- B.truelist and B.falselist are two attributes for nonterminal B, both of are lists of intermediate code statement numbers.
- B.truelist is a list of those intermediate code where B evaluates to true but the target address is not known. Each element on this list is an intermediate code whose label field for the destination is incomplete.
- B.falselist is a similar list where B evaluates to false.
- These lists contain incomplete code which contain jumps to the true and false exits of B.

The grammar complete with semantic actions is given below.

```
B → B1 or M B2
    {
        backpatch(B1.falselist, M.stat);
        B.truelist := merge(B1.truelist, B2.truelist) ;
        B.falselist := B2.falselist
    }
B → B1 and M B2    {backpatch(B1.truelist,M.stat);
                      B.falselist := merge(B1.falselist, B2.falselist) ;
                      B.truelist := B2.truelist
                      }
```

```
B → not B1
    { B.truelist := B1.falselist ; B.falselist := B1.truelist }
```

```
B → ( B1 ) { B.truelist := B1.truelist ; B.falselist := B1.falselist }
```

```
B → true { B.truelist := makelist (nextstat); emit (‘goto ___’) }
```

```
B → false { B.falselist := makelist (nextstat); emit (‘goto ___’) }
```

```
B → E1 relop E2
```

```

{ B.truelist := makelist (nextstat);
  B.falselist := makelist (nextstat + 1);
  emit ('if' E1.place 'relop' E2.place 'goto' ____);
  emit ('goto' ____);
}

```

```

B → E
{ B.place := E.place;
  B.truelist := makelist (nextstat); B.falselist := makelist (nextstat + 1);
  emit ('if' B.place 'goto' ____);
  emit ('goto' ____);
}

```

```

M → ε {M.stat := nextstat}

```

```

E → E1 + E2 | other rules
{ same actions as given earlier}

```

The attribute, M.stat, for nonterminal M is used to hold the index of the next 3-address statement at strategic points in the code.

We use the term marker nonterminal for nonterminals, such as M, which are used in this manner. The following functions and variables have also been used

1. makelist(i) is a function that creates a list containing a single element { i } and returns a pointer to the list created. The element i is an index of a 3-address statement.
2. merge(p1, p2) is a function that concatenates two lists pointed by p1 and p2 and returns a pointer to the merged list.
3. backpatch(p, i) is a function that inserts i in the place for the missing target label in each element of the list (of 3-address statements) pointed to by p.
4. nextstat is a global variable that is used to hold the index of the next 3-address statement.

The method for partial evaluation manages to generate code in a single pass over the source code.

It holds the list of incomplete statements and carries them around, along with parsing, till such time that till their target labels become available.

When the target labels are found, these are placed at the label fields of incomplete codes using the lists being carried along as semantic attributes.

The code generated is assuredly correct but may not be efficient.

Boolean expressions may be evaluated by either of the two methods. However, the choice of translation does not rest with the implementor, it is a part of the language specifications.

Partial evaluation is in some sense an efficient form of complete evaluation. However the two translations may not give identical results, particularly in the presence of side effects.

Partial evaluation is typically suited for translation of boolean expressions which are part of control flow constructs. Complete evaluation is used in the context of assignment statements.

TRANSLATION OF CONTROL FLOW CONSTRUCTS

We now take up translation of flow of control statements, such as if-then-else and while-do.

The boolean expression component of these statements are partially evaluated as explained already.

For each control flow construct, the following need to be done.

1. Determine from the context, how to resolve the true and false lists of the associated boolean expression.
2. **How to link the different components of the constructs so that the translation is semantically equivalent ?**

For example, if-then-else statement has three components, boolean expression and two statement parts.

These components have to be connected in a manner to ensure the correct flow of control and semantics as specified for this construct.

3. Three-address statements, which have jumps to outside the body of the construct, cannot be resolved at generation time.

These statements must be carried forward till the required target becomes known. Incomplete 3-address code may occur within the code generated for

- i. boolean expression,
- ii. other components of the construct, and
- iii. 3-address jump statements inserted to link the various components.

We assume, for now, that the only kinds of jump allowed from within a control flow construct is to the statement following the construct (i.e., unconditional goto is absent in the source program).

In addition to the attributes, B.truelist, B.falselist, M.stat and the functions makelist(), merge() and backpatch() and a variable nextstat the following are required.

- a) a nonterminal N , whose purpose is to prevent control flow from then part into the else part of an if-then-else statement. It causes the generation of an incomplete unconditional 3 address jump statement.

The index of this statement is held in an attribute N.nextlist.

The target jump is resolved by backpatching it with the index of the address code following the if-then-else.

b) Indices of the incomplete 3-address statements generated for S are held in an attribute S.nextlist.

c) Similarly Slist.nextlist holds a list of indices of all the incomplete 3-address code corresponding to the list of statements denoted by Slist.

$S \rightarrow \text{if } B \text{ then } M \ S_1$

```
{ backpatch(B.truelist, M.stat) ;  
  S.nextlist := merge(B.falselist, S1.nextlist)  
}
```

$S \rightarrow \text{if } B \text{ then } M_1 \ S_1 \ N \ \text{else } M_2 \ S_2$

```
{ backpatch(B.truelist, M1.stat);  
  backpatch(B.falselist, M2.stat);  
  S.nextlist := merge(S1.nextlist, merge(N.nextlist,  
                                         S2.nextlist))  
}
```

$S \rightarrow \text{while } M_1 \ B \ \text{do } M_2 \ S_1$

```
{ backpatch(B.truelist, M2.stat);  
  backpatch(S1.nextlist, M1.stat);  
  S.nextlist := B.falselist;  
  emit ('goto' M1.stat)  
}
```

$S \rightarrow \text{begin } Slist \ \text{end}$

```
{ S.nextlist := Slist.nextlist }
```

$S \rightarrow L := E$

```
{ semantic rules given earlier;  
  S.nextlist := nil  
}
```

$Slist \rightarrow Slist_1 ; M \ S$

```

{ backpatch (Slist1.nextlist, M.stat);
  Slist.nextlist := S.nextlist
}

```

```

Slist → S {Slist.nextlist := S.nextlist}

```

```

M → ε {M.stat := nextstat}

```

```

N → ε { N.nextlist := makelist (nextstat);
        emit ('goto' __)
      }

```

We illustrate the discussion with an example given below.

The symbols, \uparrow_n , are used to mark certain points of the program fragment, to illustrate generation of code, in an incremental fashion along with parsing. The marked points are clearly not part of the source program.

```

begin
  if a > b  $\uparrow_1$  then
    if b > c  $\uparrow_2$  then
      begin
        d := b + c; b := b / 2
      end  $\uparrow_3$ 
    else b := 2 * b  $\uparrow_4$ 
  else
    while a > 0  $\uparrow_5$  do
      begin

```



```

                                d := b * b; a := a / 2
                                end ↕6;
                                b := b / 2 ↕7
                                end ↕8

```

The source code has an outer if-then-else which has a nested if-then-else in its then part and a nested while loop in its else part.

It will be instructive to use the grammar over the input string and generate the intermediate code yourself along with parsing. You may assume a bottom up parser for this exercise.

The intermediate code that gets generated at the marked points, the values of semantic attributes, as parsing proceeds and semantic actions are executed are also given.

A number of stack configurations at intermediate stages of parsing are shown. The values of semantic attributes along with intermediate code fragments generated at that point of parsing show how syntax directed translation can be made to work correctly.

TRANSLATION OF CONTROL FLOW CONSTRUCTS

Assume that nextstat is 10 at start. Relevant stack configurations around the marked points are shown.

Stack Contents

Parsing & semantic Actions

Stack Configuration 1 :

begin if $a > b$

Reduction by $B \rightarrow E1 \text{ relop } E2$;
 $B.\text{truelist} = \{10\}$; $B.\text{falselist} = \{11\}$ $B \equiv a > b$

begin if $B \uparrow_1$

Code generated so far :

10 : if $a > b$ goto ____
11 : goto ____
 $B.\text{truelist} = \{10\}$; $B.\text{falselist} = \{11\}$

Stack Configuration 2:

begin if B then M_1 if $b > c$

$M_1.\text{stat} = \{12\}$;
Reduction by $B \rightarrow E1 \text{ relop } E2$;
 $B.\text{truelist} = \{12\}$; $B.\text{falselist} = \{13\}$ for $B \equiv b > c$

begin if B then M_1 if $B \uparrow_2$

Code generated so far :

10 : if $a < b$ goto ____
11 : goto ____
12 : if $b > c$ goto ____
13 : goto ____

Stack Configuration 3:

begin if B then M_1 if B then M_1 begin $d := b + c$

$M_1.\text{stat} = \{14\}$;
Reduction; $S.\text{nextlist} = \Phi$

begin if B then M_1 if B then M_1 begin S

Code generated so far :

10 : if $a < b$ goto ____	14 : $t_1 := b + c$
11 : goto ____	15 : $d := t_1$
12 : if $b > c$ goto ____	
13 : goto ____	

TRANSLATION OF CONTROL FLOW CONSTRUCTS

Stack Configuration 4:

begin if B then M_1 if B then M_1 begin Slist; $M b := b/2$

reduction; Slist.nextlist = Φ ; M.stat = {16}

begin if B then M_1 if B then M_1 begin Slist; M S

S.nextlist = Φ ; reduction

Code generated so far :

10 : if a < b goto __	14 : $t_1 := b + c$
11 : goto __	15 : $d := t_1$
12 : if b > c goto __	16 : $t_2 := b / 2$
13 : goto __	17 : $b := t_2$

Stack Configuration 5:

begin if B then M_1 if B then M_1 begin S end

S.nextlist = Φ ; reductions

Stack Configuration 6:

begin if B then M_1 if B then M_1 S₁ N \uparrow_3

N.nextlist = {18}

Code generated so far :

10 : if a < b goto __	15 : $d := t_1$
11 : goto __	16 : $t_2 := b / 2$
12 : if b > c goto __	17 : $b := t_2$
13 : goto __	18 : goto __
14 : $t_1 := b + c$	

Stack Configuration 7:

begin if B then M_1 if B then M_1 S₁ N else M_2 $b := 2 / b$

M_2 .stat = {19}; reduction

TRANSLATION OF CONTROL FLOW CONSTRUCTS

Stack Configuration 8:

begin if B then M_1 if B then M_1 S₁ N else M_2 S₂

S.nextlist = Φ ; reduction;

Stack Configuration 9:

begin if B then M_1 S₁ \uparrow_4

backpatch({12}, 14);

backpatch({13}, 19);

Code generated so far :

10 : if a < b goto ____	16 : t ₂ := b / 2
11 : goto ____	17 : b := t ₂
12 : if b > c goto 14	18 : goto ____
13 : goto 19	19: t ₃ := 2 * b
14 : t ₁ := b + c	20 : b := t ₃
15 : d := t ₁	21 : goto ____

Stack Configuration 10:

begin if B then M₁ S₁ N else M₂ while M₁ a > 0

M₂.stat = {22}; M₁.stat = {22}; reduction;

Stack Configuration 11:

begin if B then M₁ S₁ N else M₂ while M₁ B \uparrow_5

B.truelist = {22}; B.falselist = {23};

Code generated so far :

10 : if a < b goto ____	17 : b := t ₂
11 : goto ____	18 : goto ____
12 : if b > c goto 14	19: t ₃ := 2 * b
13 : goto 19	20 : b := t ₃
14 : t ₁ := b + c	21 : goto ____
15 : d := t ₁	22 : if a > 0 goto ____
16 : t ₂ := b / 2	23 : goto ____

TRANSLATION OF CONTROL FLOW CONSTRUCTS

Stack Configuration 12:

begin if B then M₁ S₁ N else M₂ while M₁ B do M₂ begin d := b * b

M₂.stat = {24}; reduction;

Code generated so far :

10 : if a < b goto ____	18 : goto ____
11 : goto ____	19: t ₃ := 2 * b
12 : if b > c goto 14	20 : b := t ₃
13 : goto 19	21 : goto ____
14 : t ₁ := b + c	22 : if a > 0 goto ____
15 : d := t ₁	23 : goto ____
16 : t ₂ := b / 2	24 : t ₄ := b * b
17 : b := t ₂	25 : d := t ₄

Stack Configuration 13:

begin if B then M_1 S_1 N else M_2 while M_1 B do M_2 begin Slist; M a := a/2

M_2 .stat = {26}; reductions; Slist.nextlist = Φ

Stack Configuration 14:

begin if B then M_1 S_1 N else M_2 while M_1 B do M_2 begin Slist; MS

Slist.nextlist = Φ ; S.nextlist = Φ

Code generated so far :

10 : if a < b goto ____	19: $t_3 := 2 * b$
11 : goto ____	20 : $b := t_3$
12 : if b > c goto 14	21 : goto ____
13 : goto 19	22 : if a > 0 goto ____
14 : $t_1 := b + c$	23 : goto ____
15 : $d := t_1$	24 : $t_4 := b * b$
16 : $t_2 := b / 2$	25 : $d := t_4$
17 : $b := t_2$	26 : $t_5 := a / 2$
18 : goto ____	27 : $a := t_5$

TRANSLATION OF CONTROL FLOW CONSTRUCTS

Stack Configuration 15:

begin if B then M_1 S_1 N else M_2 while M_1 B do M_2 begin Slist end

Slist.nextlist = Φ ; reduction; M_1 .stat = {22}

Stack Configuration 16:

begin if B then M_1 S_1 N else M_2 S_2

S_2 .nextlist = {23}; S_1 .nextlist = {18, 21};
backpatch({22}, 24); M_1 .stat = {12}
 M_2 .stat = { }; N.nextlist = {21};
B.truelist = {10}; B.falselist = {11}; reduce;

Code generated so far :

10 : if a < b goto ____	15 : $d := t_1$
11 : goto ____	16 : $t_2 := b / 2$
12 : if b > c goto 14	17 : $b := t_2$
13 : goto 19	18 : goto ____
14 : $t_1 := b + c$	19: $t_3 := 2 * b$

```

20 : b := t3
21 : goto ____
22 : if a > 0 goto 24
23 : goto ____
24 : t4 := b * b

```

```

25 : d := t4
26 : t5 := a / 2
27 : a := t5
28 : goto 22

```

Stack Configuration 17:

begin S ↑₆

```

S.nextlist = {18, 21, 23};
backpatch ({10}, 12); backpatch ({11}, 13);
reduce;

```

Code generated so far :

```

10 : if a < b goto 12
11 : goto 13
12 : if b > c goto 14
13 : goto 19
14 : t1 := b + c
15 : d := t1
16 : t2 := b / 2
17 : b := t2
18 : goto ____
19 : t3 := 2 * b

```

```

20 : b := t3
21 : goto ____
22 : if a > 0 goto 24
23 : goto ____
24 : t4 := b * b
25 : d := t4
26 : t5 := a / 2
27 : a := t5
28 : goto 22

```

Stack Configuration 18:

begin Slist ; M b := b / 2

```

Slist.nextlist = {18, 21, 23}; M.stat={29}
reduce;

```

Stack Configuration 19:

begin Slist ; M S

```

Slist.nextlist = {18, 21, 23}; M.stat={29}
S.nextlist = Φ; reduce;

```

Code generated so far :

```

10 : if a < b goto 12

```

```

11 : goto 13

```

```

22 : if a > 0 goto 24
23 : goto __
24 : t4 := b * b
25 : d := t4
26 : t5 := a / 2
27 : a := t5
28 : goto 22
29 : t6 := b / 2
30 : b := t6

```

```

21 : goto 29
22 : if a > 0 goto 24
23 : goto 29
24 : t4 := b * b
25 : d := t4
26 : t5 := a / 2
27 : a := t5
28 : goto 22
29 : t6 := b / 2
30 : b := t6

```

```

17: b := t2
18: goto 29
19: t3 := 2 * b
20: b := t3
21: goto 29
22: if a > 0 goto 24
23: goto 29

```

24 : t ₄ := b * b	28 : goto 22
25 : d := t ₄	29 : t ₆ := b / 2
26 : t ₅ := a / 2	30 : b := t ₆
27 : a := t ₅	

TRANSLATION OF OTHER CONSTRUCTS

The basic issues in semantic analysis and translation have been covered. However, a compiler has to deal with more details, a few of which are listed below.

Translation of goto statement. It is handled by storing labels in the symbol table, associating index of 3-address code with the label and using backpatching to resolve forward references.

Procedure call statement

Let parameter passing be call by reference and storage be statically allocated.

- Semantic analysis involves type checking of the formal and actual arguments.
- Translation would require generating 3-address code for evaluating each argument (in case it is an expression). Then generate a list of 3-address **param** statements, one for each argument, followed by the 3-address call statement.

Translation of other constructs such as **case** statement, **for** statement and **repeat - until** statement can be worked out on similar lines.

We have ignored semantic analysis for checking structural integrity of single-entry control structures. Code for testing such violations also need to be included.

5. Translation of Other constructs : The basic issues in semantic analysis and translation have been covered. However, a compiler has to deal with more details, a few of which are listed below. Translation of goto statement or break statements. It is handled by storing labels in the symbol table, associating index of 3-address code with the label and using backpatching to resolve forward references.

Procedure call statement

Consider a function definition :

```
void func (int a, int * p, float f)
{ /* body skipped */ return; }
```

and a corresponding call to func() as given below.

```
func(i*j+5, &q+5, 25 *pi); // i, j are int; q is int*; and pi is float)
```

the desired intermediate code is. Note how each actual argument is evaluated into a temporary of the same type as that defined in the corresponding formal argument. The use of param statements, one for each actual parameter followed by a call to the function.

10: t1 := i * j	17: t8 := intoreal t6
11: t2 := t1 + 5	18: t9 := t8*pi
12: t3 := &q	19: param t2
13: t4 := 5*4	19 : param t6
14: t5 := t3+t4	20 : param t9
15: t6 := &t5	21 : call func
16 : t7 = 25	

Parameter passing may be call by value or call by reference and are processed accordingly..

- a) Semantic analysis involves type checking of the formal and actual arguments.
- b) Translation would require generating 3-address code for evaluating each argument (in case it is an expression). Then generate a list of 3-address **param** statements, one for each argument, followed by the 3-address call statement.

Translation of other constructs such as **case** statement, can be worked out on similar lines.

RUN TIME ENVIRONMENTS IN COMPILER DESIGN

Supratim Biswas

Retired Professor, IITB & Visiting Professor BITM

sb@cse.iitb.ac.in

supratim.biswas@bitmesra.ac.in

Course Material (Revised Version Designed in 2018)

Re-Revised Version, April 2024 for CS333

April 2024



Department of Computer Science & Engineering

Indian Institute of Technology, Bombay

Birla Institute of Technology, MESRA

We shall give a quick review of the following before discussing the basic issues in the domain of runtime environments.

1. Intermediate Code Form

2. Architecture of x86 – 64 bit architecture : a brief review. The run time environment is better explained using examples from a standard architecture and this is the architecture which we are using for our experimentation in the laboratory.

Readers who are familiar with both of above may skip and directly go to Section 3 on Page 8.

1. INTERMEDIATE CODE FORM

The front end typically outputs an explicit form of the source program for subsequent analysis by the back end.

The semantic actions incorporated along with the grammar rules are responsible for emitting them.

Among the several existing intermediate code forms, we shall consider one that is known as Three-Address Code.

In three address code, each statement contains 3 addresses, two for the operands and one for the result. The form is similar to assembly code and such statements may have a symbolic label.

The commonly used three address statements are given below.

1. Assignment statements of the form :
 $x := y \text{ op } z$, where op is binary, or
 $x := \text{op } y$, when op is unary
2. Copy statements of the form $x := y$
3. Unconditional jumps, goto L. The three address code with label L is the next instruction where control flows.
4. Conditional jumps, such as
 if $x \text{ relop } y$ goto L
 if x goto L

The first instruction applies a relational operator, relop, to x and y and executes statement with label L, if the relation is true. Else the statement following if $x \text{ relop } y$ goto L is executed. The semantics of the other one is similar.

5. For procedure calls, this language provides the following :

param x

call p,n

where n indicates the number of parameters in the call of p.

6. For handling arrays and indexed statements, it supports statements of the form :

$x := y[i]$

$x[i] := y$

The first is used to assign to x the value in the location that is i units beyond location y, where x, y and i refer to data objects.

7. For pointer and address assignments, it has the statements

$x := \& y$

$x := * y$

$* x := y .$

In the first form, y should be an object (perhaps an expression) that admits a l-value. In the second one, y is a pointer whose r-value is a location. The last sets r-value of the object pointed to by x to the r-value of y.

A three address code is an abstract form of intermediate code. It can be realised in several ways, one of them is called **Quadruples**.

A quadruple is a record structure with 4 fields, usually denoted by op, arg1, arg2 and result. For example, the quadruples for the expression $a + b * c$ is

arg1	arg2	result	op	Equivalent form	
*	b	c		t_1	$t_1 = b * c$
+	a	t_1		t_2	$t_2 = a + t_1$

where t_1 and t_2 are compiler generated temporaries.

2. X86-64 bit Architecture at a Glance

- x86 assembly code has been popular for a long time.
- With replacement of 32-bit PCs with 64-bit ones, the assembly code has changed.
- x64 is a generic name for the 64-bit extensions to Intel and AMD architectures.

Registers :

16 general purpose 64-bit registers, the first 8 are named as RAX, RBX, RCX, RDX, RBP, RSI, RDI, and RSP.

- The second eight are named R8-R15. By replacing the initial R with an E on the first eight registers, the lower 32 bits can be accessed. For instance, EAX for RAX and similar for RAX, RBX, RCX, and RDX. The lower 16 bits are also accessible by dropping the prefix R, such as AX for RAX, etc.
- The lower byte of these registers are accessible by replacing the X for L, such as AL for AX. Similarly for the higher byte of the low 16 bits of X, such as AH for AX.
- The new registers R8 to R15 can be accessed in a similar manner, such as : R8 (qword), R8D (lower dword), R8W (lowest word), R8B or R8L. Note there is no R8H.

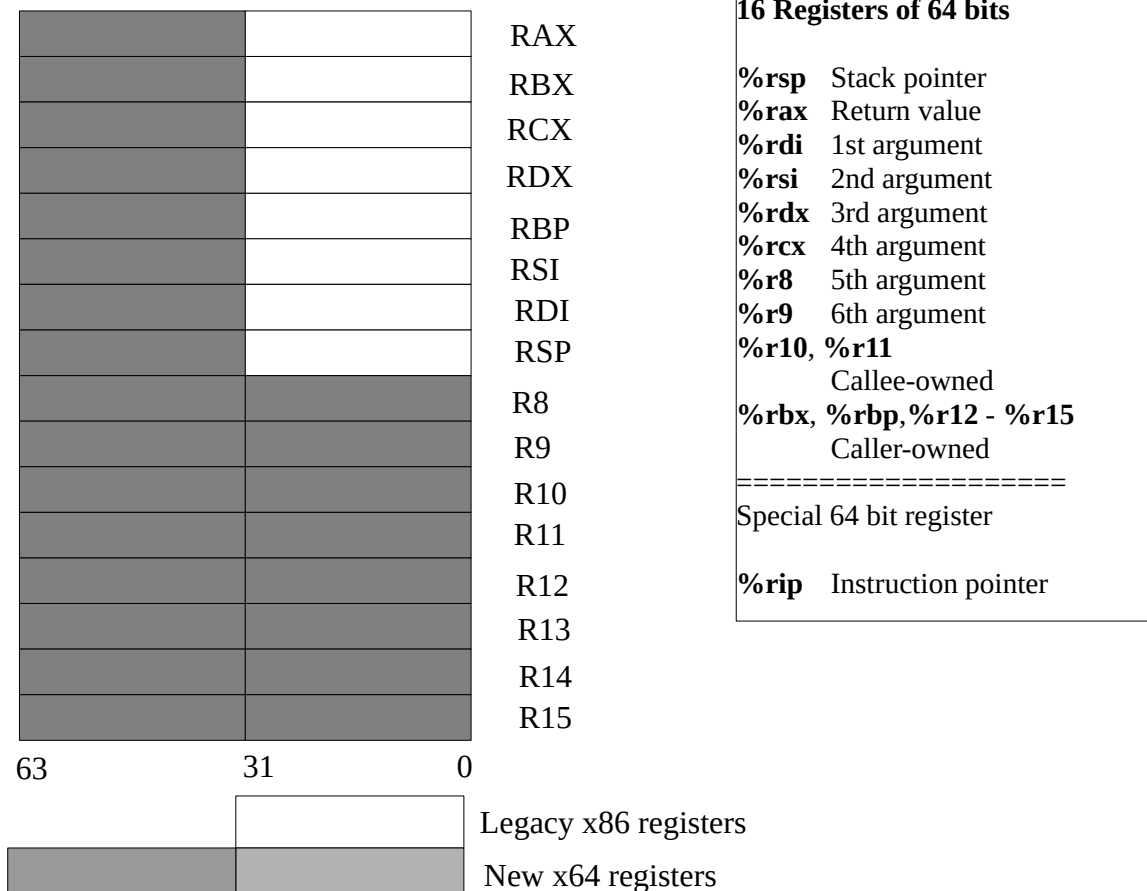
RIP is a 64-bit instruction pointer which points to the next instruction to be executed.

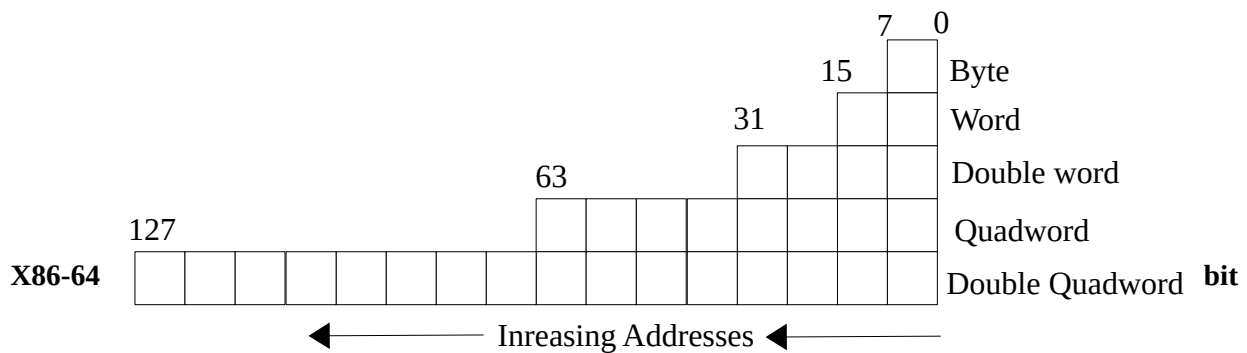
The stack pointer RSP points to the last item pushed onto the stack, which grows toward lower addresses.

The stack is used to store return addresses for functions, for passing parameters, saving registers, etc.

The registers and their usage conventions are listed here.

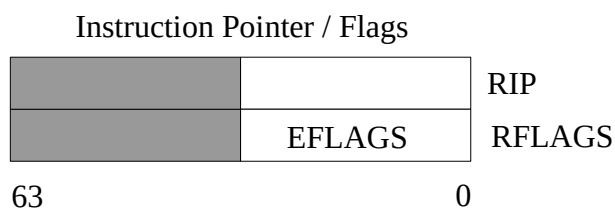
X86-64 bit Architecture





Architecture

The RFLAGS register stores flags used for results of operations and for controlling the processor. The useful flags are listed below.



Condition codes / Flags

CF	Carry Flag
OF	Overflow Flag
SF	Sign Flag
ZF	Zero Flag

Common Flags

Symbol	Bit	Name	Set when
CF	0	Carry	Operation generated a carry or a borrow
PF	2	Parity	Last byte has even number of 1's, else 0
ZF	6	Zero	Result was 0
SF	7	Sign	Most significant bit of result is 1
OF	11	Overflow	Overflow on signed operation

Common Assembly Instructions

Code	Operands	Semantics
mov	src, dst	# dst = src
movsbl	src, dst	# byte to int, signl-extend
movzbl	src, dst	
lea	addr, dst	# dst = addr
add	src, dst	# dst += src
sub	src, dst	# dst -= src
imul	src, dst	# dst *= src
neg	dst	# dst = - dst (arith inverse)
sal	count, dst	# dst <<= count (arith shift)
sar	count, dst	# dst >>= count (arith shift)
shr	count, dst	# dst >>= count (logical shift)

and	src, dst	# dst &= src
or	src, dst	# dst = src
Code	Operands	Semantics
xor	src, dst	# dst ^= src
not	dst	# dst = ~dst (bitwise inverse)
jmp	label	# unconditional jump to label
je	label	# jump on equal; ZF=1
jne	label	# jump on not equal; ZF=0
js	label	# jump on negative; SF=1
jns	label	# jump on not negative; SF=0
jg	label	# jump on > (signed); ZF=0 and SF=OF
jge	label	# jump on >= (signed); SF=OF
jl	label	# jump on < (signed); SF!=OF
jle	label	# jump on <= (signed); ZF=1 or SF!=OF
ja	label	# jump on > (unsigned); CF=0 and ZF=0
jb	label	# jump on < (unsigned); CF=1
cmp	a, b	# b-a, set flags
test	a, b	# a&b, set flags
push	src	# add to top of stack; Mem[--%rsp] = src
pop	dst	# remove top from stack; # dst = Mem[%rsp++]
call fn		# push %rip, jmp to fn
ret		# pop %rip

Instruction suffixes

b	byte
w	word (2 bytes)
l	long /doubleword (4 bytes)
q	quadword (8 bytes)

Suffix is omitted when it can be inferred from operands, %rax implies q, %eax implies l, etc.

Addressing modes : Operands to MOV instruction

Immediate

mov \$0x5, dst

\$val : source is **constant value**

Register

mov %rax, dst

%R: R is register; source in **%R**

Direct

mov 0x4033d0, dst

0xaddr : source read from **Mem[0xaddr]**

Indirect

mov (%rax), dst

(%R) : R is register; source read from **Mem[%R]**

Indirect displacement

mov 8(%rax), dst

D(%R); R is reg; D is displacement; source from **Mem[%R + D]**

Indirect scaled-index

mov 8(%rsp, %rcx, 4), dst

D(%RB,%RI,S); RB is register for base

RI is register for index (0 if empty); D is displacement (0 if empty)

S is scale 1, 2, 4 or 8 (1 if empty);

source from **Mem[%RB + D + S*%RI]**

3. Understanding the Key Tasks performed by a Compiler under Run Time Environment (RTE)

The compiler on completion of syntax and semantic analysis, has in a broad sense done the following two key tasks.

- Constructed a linked symbol table for all features that can define symbols in that language.
- Verified that the use of every symbol is valid and resolved this use to exactly one definition of that symbol. In case there are errors in definition or use of a symbol, error messages(s) are communicated and no further semantic analysis of the remaining code is done. However the compiler continues to perform syntax analysis of the remaining program, after the error point, to discover and report syntax errors therein.
- Generated intermediate code, wherever relevant, and has saved intermediate code for the entire source program, for further processing. The syntax and semantics of intermediate code, that is being used in this course, are reproduced in a separate section.
- The tasks of the compiler after successful syntax and semantic analyses, are
 - Optimization of the intermediate code, if this option is invoked by the user, and
 - Code generation for the target architecture. In this course, we assume the target code to be the assembly language of a given architecture, namely x86 64 bit architecture. A quick review of this architecture is included as a separate section.
- Run time environment comprises of a series of activities undertaken by the compiler to prepare the intermediate code towards generation of assembly code.
 - Use the symbol table data to decide physical memory for data in the program.
 - Use the symbol table to decide the layout of memory for keeping the assembled code of all functions of the program.
 - Based on the compiler decisions of memory layout of data and code fragments, change the references to code and data in the intermediate code of the functions to correspond to the assigned layout in the assembly code.
 - Function **call** and **return** require special handling to deal with the context changes when the transfer of control happens
 - from the code of the caller function (the function that makes a call) to the callee function (the function that is called) at the point of call in the body of the caller.
 - From callee function to its caller, when a return is encountered in the body of the callee.

These notes address all the issues mentioned above with illustrative examples. Note that

- we have chosen to use the combination of C language as source and X86 64 bit architecture as the target to explicate the issues in a concrete form without any guesses. This approach is expected to demystify this often underrated task of a compiler, and also provide good grasp about the generated assembly code.

- The RTE issues and solutions discussed in this document are however more general than the customization for C and x86 64bit architecture combination. The generic issues are pointed at relevant places in this document.
- With minor changes in the framework, the solutions discussed can be adapted to other combination pairs of (language, m/c). However one needs to know the language features and specifications for data and code and for function call and return.

Example 1 : Explore thorough experimentation the layout of data and code as done by gcc compiler for such features present in programs.

Source program “mem-layout.c”	Output on execution
<pre>int c = 4; static float val = 5.0; float f(float x) { return x*x - 2.0;} int main() { int a, b; a = a + b + c; int (*pm) () = &main; printf("&c = %p &a = %p &b = %p &val = %p\n", &c, &a, &b, &val); printf(" &main() %p \n", pm); printf(" f(val) = %f \n", f(val)); return 0; }</pre>	<pre>c : int global val : static float f : function with x : float argument returns float main : function returns int &c = 0x55944a1e2010 &a = 0x7ffc91b4ca88 &b = 0x7ffc91b4ca8c &val = 0x55944a1e2014 &main() 0x55944a1df18d f(val) = 23.000000</pre>

Let us interpret the addresses assigned by the compiler to data and code. The addresses when sorted in ascending order are (leading prefixes 0x are removed, since we know that all addresses are in hex representation only). The layout as done by gcc is depicted below.



The following may be noted in this context.

1. The code for main() occupies the lowest address among the rest. This is the starting address of the assembly code for main(). The right end of the address of main() will be decided by the size of code of main() [**Q. can you find out the size of main()’s assembly code in bytes ?**].
2. The global data c and the static data val reside at adjacent locations at a certain offset from the start address of main(). Note the gap between the start address of main() and c is “2e83” in hex which is quite close. **Q. Is the small gap accidental or intentional ?**
3. The gap in the memory between address of static **val** and local **a** of main() is huge.

The observations noted above are true for most languages. The following is the recommended memory layout for code / data at a generic level across levels and architectures.

INITIAL DIVISION OF MEMORY

The memory is usually divided into 4 parts initially.

1. **Code area**: Contains the generated target code, for example the code for functions, `f()`, `g()`, `main()`, etc.
2. **Static area** : Contains data whose absolute addresses can be determined at compile time. For example,
 - In C the addresses of globals, static data are kept in static area.
 - In languages like Pascal, the addresses of the local variables of the outermost procedure can be determined at compile time.
 - In language like Fortran IV, the addresses of all variables can be determined statically (recursion not permitted).
3. **Stack area** : For data objects of procedures which can have more than one incarnations active at the same time. The local data of a recursive function is an example.
4. **Heap area** : This is required for dynamically allocated data (like objects pointed to by pointer types). Since the sizes of the stack and the heap are not known at compile time, they are arranged to grow in opposite directions for effective utilization of space allocated to stack and heap.



The gaps between the different areas are also intentional.

- Keeping the code and global data in close proximity permits the compiler to share the instruction pointer, `%rip` for our architecture, (also known as program counter PC). The instruction pointer register, `%rip`, is a dedicated register used to hold the address of the next instruction to be executed in the code of the currently executing function.
- By allocating memory close to the code area, `%rip` may be used to access the globals by using the offset of the global from the start of code area. This is a read only use of `%rip` so it does not affect the basic role of `%rip`.
- The huge gap between {code, global} and local variables of a function is necessitated by the need to support functions with large body and global data of large sizes. Function call and return, specially recursive functions, are realized using a stack. All contemporary architectures provide a stack implemented in the hardware, to support recursive functions and associated issues related to parameter passing, local data, call and return.
- Most Languages of today, including C, support dynamically allocated data. The area of memory that is allocated for this purpose, to realize `malloc()`, `free()` and related functions, is known as a **Heap**. Since our sample example did not use dynamic data, address of an element in **heap memory** is not seen here. You can extend the given program to create a dynamic data and find out the part of memory that gcc uses to layout the heap.

4. ILLUSTRATION OF COMPILATION ISSUES THROUGH EXAMPLES

Example 1 – Simple Program in C

```
int a,b;           // global data
int main ()
{
    a = a + b;
}
```

\$ gcc -S -fverbose-asm run1.c

<pre>.file "run1.c" .globl a .data .align 4 .type a, @object .size a, 4 a: .long 4 .globl b .align 4 .type b, @object .size b, 4 b: .long 16 .text</pre>	<pre>.globl main .type main, @function main: .LFB0: pushq %rbp # movq %rsp, %rbp #, movl a(%rip), %edx # a movl b(%rip), %eax # b addl %edx, %eax # movl %eax, a(%rip) # %eax, a movl \$0, %eax # popq %rbp # ret Both a and b are given absolute addresses</pre>
--	---

Note that `%rip` is the instruction pointer; `a(%rip)` and `b(%rip)` show that the addresses of a and b are with respect to `%rip`; they are in the code area and hence absolute addresses.

Example 2 – Function with local variables

Source C Program : run2.c	Relevant Assembly Code
<pre>void f() { int a, b; // local variables to f() a = a + b; }</pre>	<pre>.file "run2.c" f: .LFB0: pushq %rbp # movq %rsp, %rbp #, movl -4(%rbp), %eax # b &b = %rbp - 4 addl %eax, -8(%rbp) # a &a = %rbp - 8; a = a + b nop popq %rbp # ret</pre>

a and b have been given relative address in the stack using %rbp

Note that %rbp is the base register that is a pointer to the stack denoting the area on stack allocated for a function.

The contents on the stack between %rbp and %rsp is the area allocated during the execution of a function call.

Example 3 : Function using both local and global data

Source program “run3.c”	Assembly code (relevant part)
<pre>#include <stdio.h> int a = 10, c; void f() { int a, b; a = a + b + c; }</pre>	<pre>.file "run3.c" f: .LFB0: endbr64 pushq %rbp movq %rsp, %rbp movl -8(%rbp), %edx movl -4(%rbp), %eax addl %eax, %edx movl c(%rip), %eax addl %edx, %eax movl %eax, -8(%rbp) nop popq %rbp ret</pre>

a and b have been given relative address in the stack using %rbp, while c has absolute address

Example 4 : Function using local and parameters

Source program "run4.c"	Assembly code (relevant part)
<pre>#include <stdio.h> void f(int x) { int a, b; a = b + x; }</pre>	<pre>.file "run4.c" .text f: .LFB0: endbr64 pushq %rbp movq %rsp, %rbp movl %edi, -20(%rbp) ## param x movl -8(%rbp), %edx ## b movl -20(%rbp), %eax addl %edx, %eax movl %eax, -4(%rbp) ## a nop popq %rbp ret</pre>

Example 5 : Function with multiple parameters and local variables

Source program "run5.c"	Assembly code (relevant part)
<pre>void f(int x, int y, int z, int*p) { int a, b; a = b + x + y + z + *p; }</pre>	<pre>.file "run5.c" f: .LFB0: endbr64 pushq %rbp # movq %rsp, %rbp #, movl %edi, -20(%rbp) # x, x movl %esi, -24(%rbp) # y, y movl %edx, -28(%rbp) # z, z movq %rcx, -40(%rbp) # p, p movl -8(%rbp), %edx # b, tmp86 movl -20(%rbp), %eax # x, tmp87 addl %eax, %edx # tmp87, _1 movl -24(%rbp), %eax # y, tmp88 addl %eax, %edx # tmp88, _2 movl -28(%rbp), %eax # z, tmp89 addl %eax, %edx # tmp89, _3 movq -40(%rbp), %rax # p, tmp90 movl (%rax), %eax # *p_10(D), _4 addl %edx, %eax # _3, tmp91 movl %eax, -4(%rbp) # tmp91, a nop popq %rbp ret .globl main</pre>

Source program “run5.c”	Assembly code (relevant part)
<pre> int main() { int n=10, m=11, p=12; int *q = &p; f(n, m, p, q); return 0; } </pre>	<pre> .type main, @function main: .LFB1: endbr64 pushq %rbp # movq %rsp, %rbp #, subq \$32, %rsp #, movq %fs:40, %rax # movq %rax, -8(%rbp) # tmp90, D.2333 xorl %eax, %eax # tmp90 movl \$10, -24(%rbp) #, n movl \$11, -20(%rbp) #, m movl \$12, -28(%rbp) #, p leaq -28(%rbp), %rax #, tmp85 movq %rax, -16(%rbp) # tmp85, q movl -28(%rbp), %edx # p, p.0_1 movq -16(%rbp), %rcx # q, tmp86 movl -20(%rbp), %esi # m, tmp87 movl -24(%rbp), %eax # n, tmp88 movl %eax, %edi # tmp88, call f # movl \$0, %eax #, _8 movq -8(%rbp), %rdi # D.2333, tmp91 xorq %fs:40, %rdi leave ret </pre>

Issue : **Q. How does f() refer to its formal arguments ? Q. Who supplies the actual argument to f() and how ?**

Example 6 : Global and Local Data and Parameters

Source program “run6.c”	Assembly code (relevant part)
<pre> #include <stdio.h> int a; short b; float x[16]; double y[20]; void f(int z) { int x, y; x = x + y; z = x + z; a = a + b; b = b + z; } </pre>	<pre> .file "run6.c" f: .LFB0: endbr64 pushq %rbp # movq %rsp, %rbp #, movl %edi, -20(%rbp) # z, z movl -4(%rbp), %eax # y, tmp91 addl %eax, -8(%rbp) # tmp91, x movl -8(%rbp), %eax # x, tmp92 addl %eax, -20(%rbp) # tmp92, z movzwl b(%rip), %eax # b, b.0_1 movswl %ax, %edx # b.0_1, _2 movl a(%rip), %eax # a, a.1_3 addl %edx, %eax # _2, _4 movl %eax, a(%rip) # _4, a movl -20(%rbp), %eax # z, tmp93 movl %eax, %edx # tmp93, _5 movzwl b(%rip), %eax # b, b.2_6 addl %edx, %eax # _5, _8 movw %ax, b(%rip) # _9, b nop popq %rbp # ret </pre>

Example 7 : Accessing array variables

Source C program	Assembly Code
<pre>void p() { int a[6], b, i; for (i = 0; i < 6; i++) b = b + a[i]; }</pre>	<pre>p: .LFB0: pushq %rbp # movq %rsp, %rbp #, subq \$48, %rsp #, movq %fs:40, %rax # movq %rax, -8(%rbp) # xorl %eax, %eax # movl \$0, -36(%rbp) # i jmp .L2 # .L3: movl -36(%rbp), %eax # i cltq movl -32(%rbp,%rax,4), %eax addl %eax, -40(%rbp) # b addl \$1, -36(%rbp) #, i .L2: cmpl \$5, -36(%rbp) # i jle .L3 # nop movq -8(%rbp), %rax xorq %fs:40, %rax # je .L4 #, .L4: leave ret</pre>

Note that p() does not have any arguments. Note how the loop has been translated.

1. Initialization $i = 0$: `movl $0, -36(%rbp) # i`
2. Test condition with code at .L2
3. Body of the loop starts at label .L3

Example 8 : Recursive Function

Source program “run7.c”	Assembly code (relevant part)
<pre>int f(int x) { int a; if (x == 0) return 1; { a = f(x-1); return (x*a); } }</pre>	<pre>.file "run8.c" f: .LFB0: endbr64 pushq %rbp # movq %rsp, %rbp #, subq \$32, %rsp #, movl %edi, -20(%rbp) # x, x cmpl \$0, -20(%rbp) #, x jne .L2 #, movl \$1, %eax #, _2 jmp .L3 # .L2: movl -20(%rbp), %eax # x, tmp85 subl \$1, %eax #, _1 movl %eax, %edi # _1, call f # movl %eax, -4(%rbp) # tmp86, a movl -20(%rbp), %eax # x, tmp87 imull -4(%rbp), %eax # a, _2 .L3: leave ret .LC0: .string " n = %d n! = %d \n" .text .globl main .type main, @function main: .LFB1: endbr64 pushq %rbp # movq %rsp, %rbp #, subq \$16, %rsp #, movl \$7, -4(%rbp) #, num movl -4(%rbp), %eax # num, tmp85 movl %eax, %edi # tmp85, call f # movl %eax, %edx #, _1 movl -4(%rbp), %eax # num, tmp86 movl %eax, %esi # tmp86, leaq .LC0(%rip), %rdi#, movl \$0, %eax #, call printf@PLT # movl \$0, %eax #, _6 leave ret</pre>

Q. Identify the compilation of the call statement in the source:

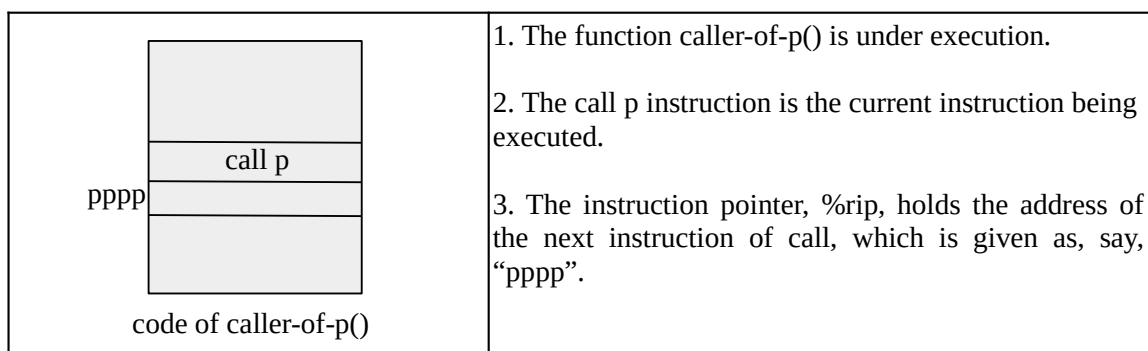
Example 9 : The program fragment in this example, has a global struct definition followed by a function p(). The function declares an object x of the struct rec and a pointer y to struct rec. The assembly code, after stripping off parts that are not directly relevant is shown in column 2.

Source program “run8.c”	Assembly code (relevant part)
<pre>#include <stdio.h> #include <stdlib.h> typedef struct list { int data; struct list* next; }rec; void p() { rec x; rec*y; y = (rec*) malloc(sizeof(rec*)); x.next = y; }</pre>	<pre>.file "run8.c" p: .LFB6: endbr64 pushq %rbp # movq %rsp, %rbp #, subq \$32, %rsp #, movl \$8, %edi # arg 1 call malloc@PLT # movq %rax, -24(%rbp) # tmp82, y movq -24(%rbp), %rax # y, tmp83 movq %rax, -8(%rbp) # tmp83, x.next nop leave ret</pre>

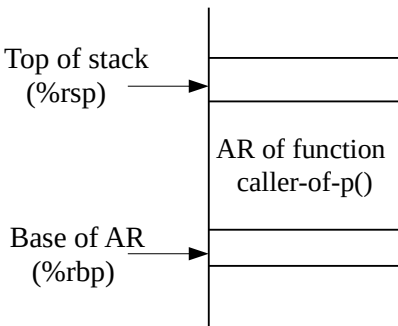
We have seen the data layout for various data that are defined and referred to in a program. However for complete understanding of the procedure call and return mechanism, we shall open up the execution of the given source through a blow by blow account as the execution proceeds at run time. However, it may be noted that while actions are taken at run time, the code that executes at run time have been carefully inserted by the compiler at the compilation time itself. The ensuing discussion should be helpful in understanding this non-trivial task performed by the compiler.

The purpose and presence of the run time environment related aspects in the assembly code are explained below. We make the following assumptions to complete the story.

- There will some function that calls p(), let us name it the “caller-of-p()”. Execution control gets transferred to p() from “caller-of-p()”, because of a call statement in the body of “caller-of-p()”. A possible scenario for this to happen is drawn below.



- The compiler allocates a certain amount of space on the stack for a function to deal with its data and other associated information. The extent of space allocated depends on the size of its local data, parameters and such information. In fact every information of a function, except its code, are on the stack. This space on the stack is referred to as the Activation Record (AR) of the function.

AR of caller-of-p()	
 <p>Top of stack (%rsp) →</p> <p>Base of AR (%rbp) →</p> <p>AR of function caller-of-p()</p>	<p>The AR of a function is addressed by two separate pointers,</p> <ol style="list-style-type: none"> 1. base pointer of the function in AR, usually kept in a dedicated register, for us it is %rbp 2. The other extent of the AR is denoted by another dedicated register, which is %rsp for us. 3. While most information related to a function lies in the AR region between its %rbp and %rsp, a function may access the stack area below its AR also, if required, depending on the communication protocol between the caller and callee functions.

- The AR is created when a function call is made and remains in force till the call is active. When the function returns from the call, its space on the AR is no longer valid, and for all practical purposes, it may be assumed that this space has been deallocated and no longer accessible.
- The division of responsibility between the functions, the caller and the callee, has to be clearly specified in order to take care of the multiple tasks, such as
 - creation of AR
 - transfer control from caller to callee
 - enable passing of parameters from caller to callee
 - access return value sent from callee to caller
 - transfer of control from callee back to caller

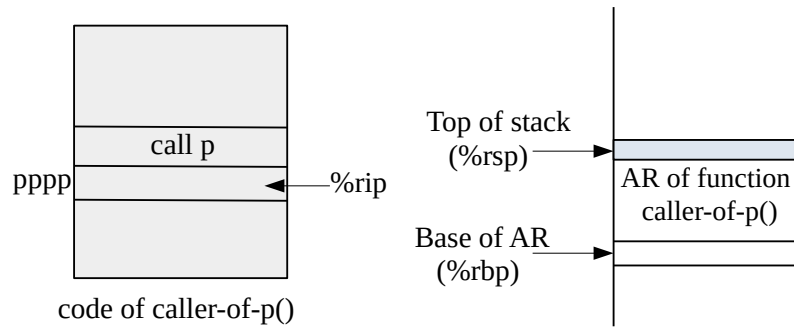
The division depends primarily on the protocols defined and agreed between designers of architecture and compiler writers.

- We study the protocol for x86 64 bit architecture and C language designers / compiler writers. The approach taken by us is to understand how gcc performs this task in complete detail. Therefore we examine assembly code generated by gcc for various simple C program fragments so as to comprehend the issues involved.
- The call instruction involves the following two actions.

call p	<ol style="list-style-type: none"> 1. Save the return address in the code of the caller, which is the address of the instruction immediately after the call in the AR 2. Transfer control from caller-of-p() to the first instruction in body of p() 	<ol style="list-style-type: none"> 1. push %rip 2. jmp @p
--------	--	---

- Let us trace the actions that are taken at execution time from the point the call is made from within the body of caller-to-p() to p() and continue till the call from p() returns to its caller.

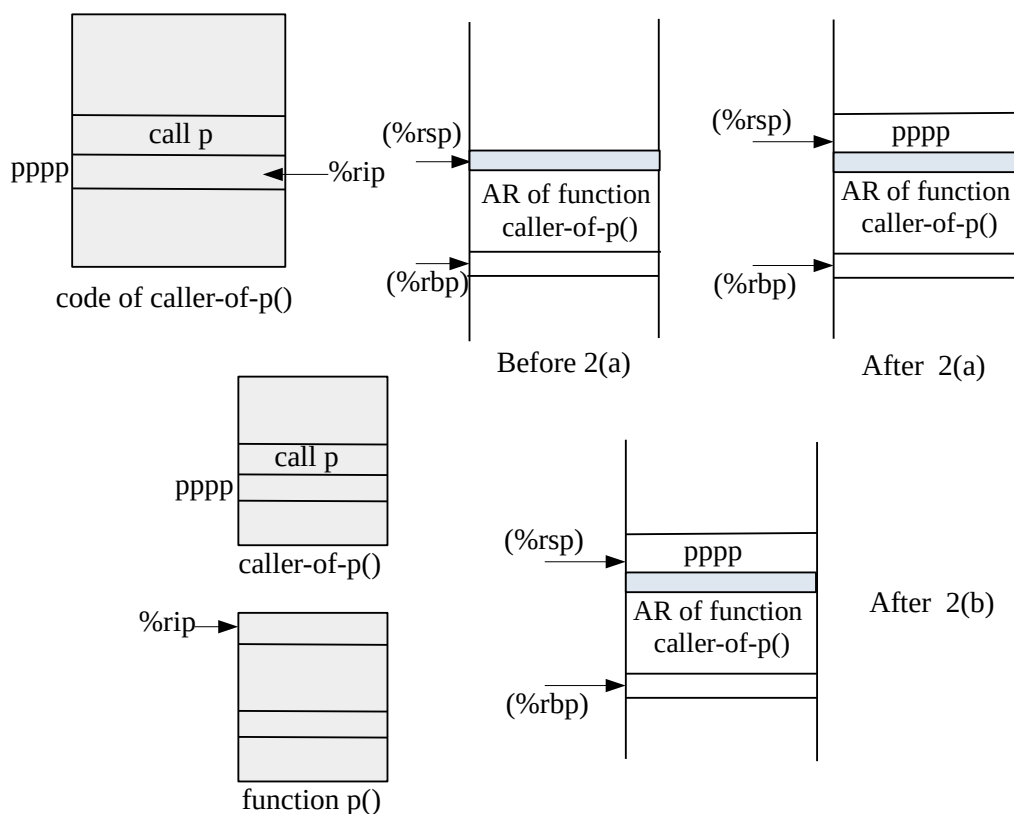
1. The starting point is when the call instruction is getting executed in caller-to-p().



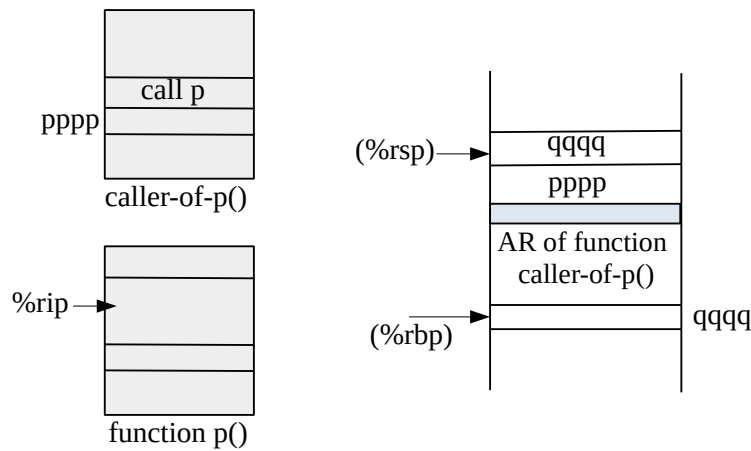
2. (a) The first part of call instruction is to save the return address on the stack, for our architecture the instruction is : pushq %rip

(b) The 2nd part is to transfer control to the first instruction of the callee function, p(); this is achieved by jmp @p

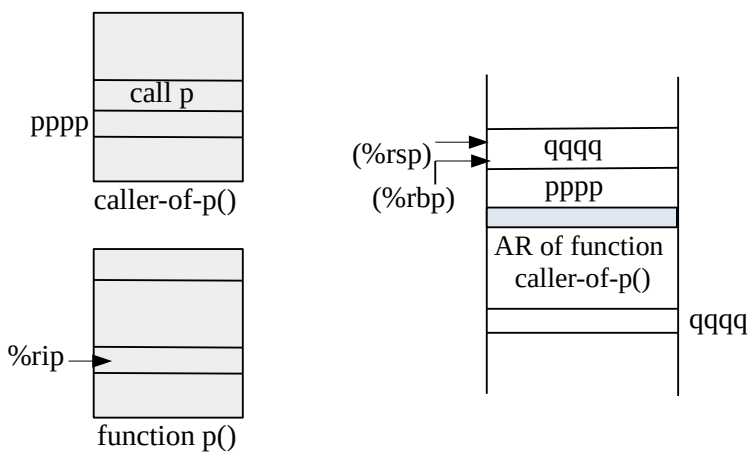
The changes in the code area and ARs on the stack after execution of each instruction is shown in the figures that follow.



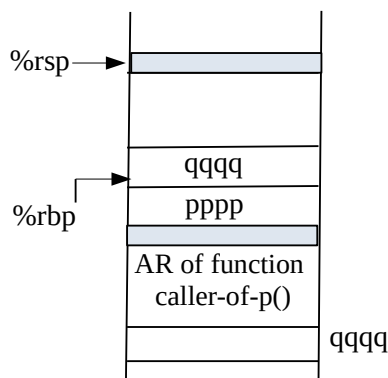
3. Since the `%rip` has been moved to the code of `p()`, the 1st instruction in the code of `p()`, that is “`endbr64`” is executed. This instruction is introduced by Intel as a security measure and hence skipped. The next instruction, “`pushq %rbp`” is then executed. The changed contents are shown below.



4. The next instruction in `p()` is “`movq %rsp, %rbp`”. The changed configuration follows.



5. The instruction to be executed at this point is : “`subq $32, %rsp`”. In our architecture, the stack grows from higher addresses to low addresses, hence this instruction moves `%rsp` by 32 bytes upward. The objective is to create space for locals of `p()`, as shown below by depicting the stack only.



6. “movl \$8, %edi” is the next instruction. The 1st executable statement in the source of p() and the corresponding assembly instruction are highlighted in the following table. The size of a pointer is 8 bytes and the compiler is preparing to call malloc(8).

Source program “run8.c”	Assembly code (relevant part)
<pre>void p() { rec x; rec*y; y = (rec*) malloc(sizeof(rec*)); x.next = y; }</pre>	<pre>movl \$8, %edi # arg 1 call malloc@PLT # movq %rax, -24(%rbp) # tmp82, y movq -24(%rbp), %rax # y, tmp83 movq %rax, -8(%rbp) # tmp83, x.next nop leave ret</pre>

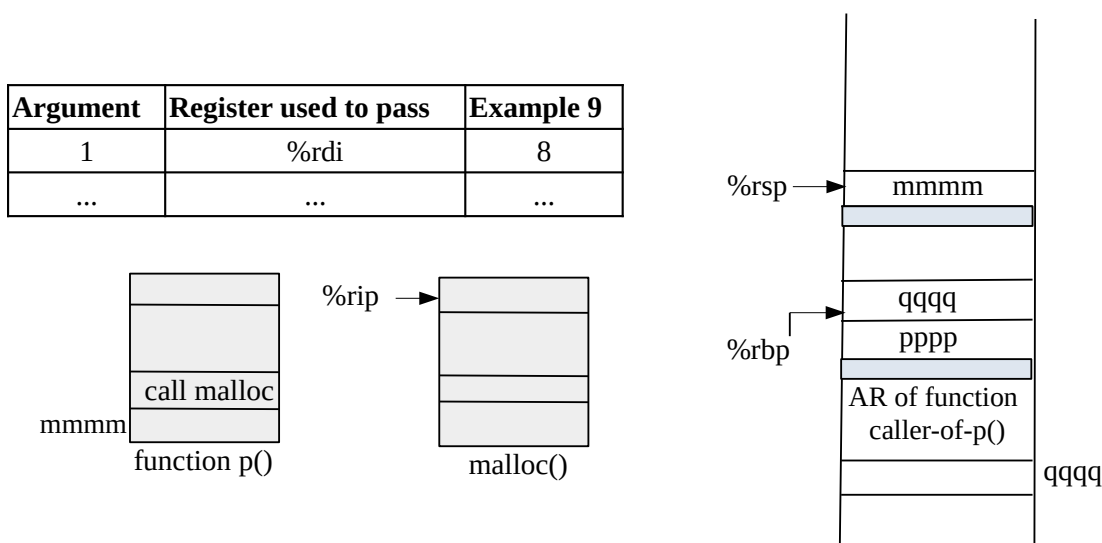
In the intermediate code this call statement in C would be translated to

C Source	IC	Assembly
y = (rec*) malloc(sizeof(rec*));	param 8 call malloc	movl \$8, %edi # arg 1 call malloc@PLT #

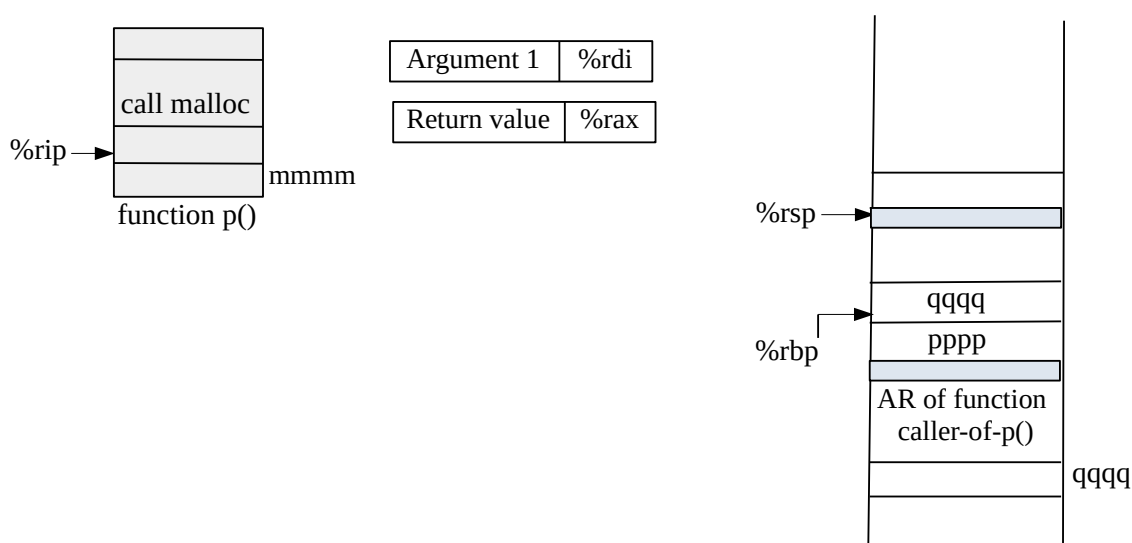
Recall argument passing conventions in our architecture : first 6 actual arguments of the callee function are passed by the caller using 6 specific registers. If the callee has more than 6 arguments, the the caller places the remaining arguments on the stack for the callee to access them from its AR. The use of %rdi to pass first argument, value 8, is consistent with this architecture.

Argument	Register used to pass	Example 9
1	%rdi	8
2	%rsi	
3	%rdx	
4	%rcx	
5	%r8	
6	%r9	

7. After setting up the lone actual argument, p() now calls malloc(), which is a library call in C, the call is serviced using definitions present in “stdlib.h”. The changed configuration follows.



The code of malloc(), though not seen by us, because of calling convention, has the same protocol. However since our focus is on the body of p(), the scene immediately after malloc() returns control back to p() needs to be drawn.



8. In the above configuration, p() executes the instruction : **movq %rax, -24(%rbp)**. We need some clarification about the operand “-24 (%rbp)”. It should be obvious that this area is in the AR of p(), since %rsp was moved by -32 in an earlier instruction. The immediate question is about the local variables of p(), namely, rec x and rec*y. The object x needs 12 bytes to hold an int and pointer, while y needs 8 bytes. The order of the locals are : {x.data, x.next, y} and they have to be in the AR of p() on the stack.

It is easy to see from the assembly code that address(x.next) is -8(%rbp) and -24(%rbp) respectively. Though not given in the assembly code, because we have not accessed x.data, address(x.data) is -4(%rbp). These details can be now be shown on the AR of p(). Also the purpose of the instruction of “movq %rax, -24(%rbp)” is to save the return value of malloc(), which is a 8 byte pointer, in the location of y on the stack. The gaps in space that we often find between consecutive data on

stack is because of padding used by the compiler to use double word address boundaries to access data.

Reader : Draw the configuration here with relative addresses of locals.

9. The next useful instruction is “leave”.

leave	1.Set the stack pointer to that of caller 2. Set the base pointer to that of caller function	1. movl %rbp, %rsp 2. popq %rbp
--------------	---	------------------------------------

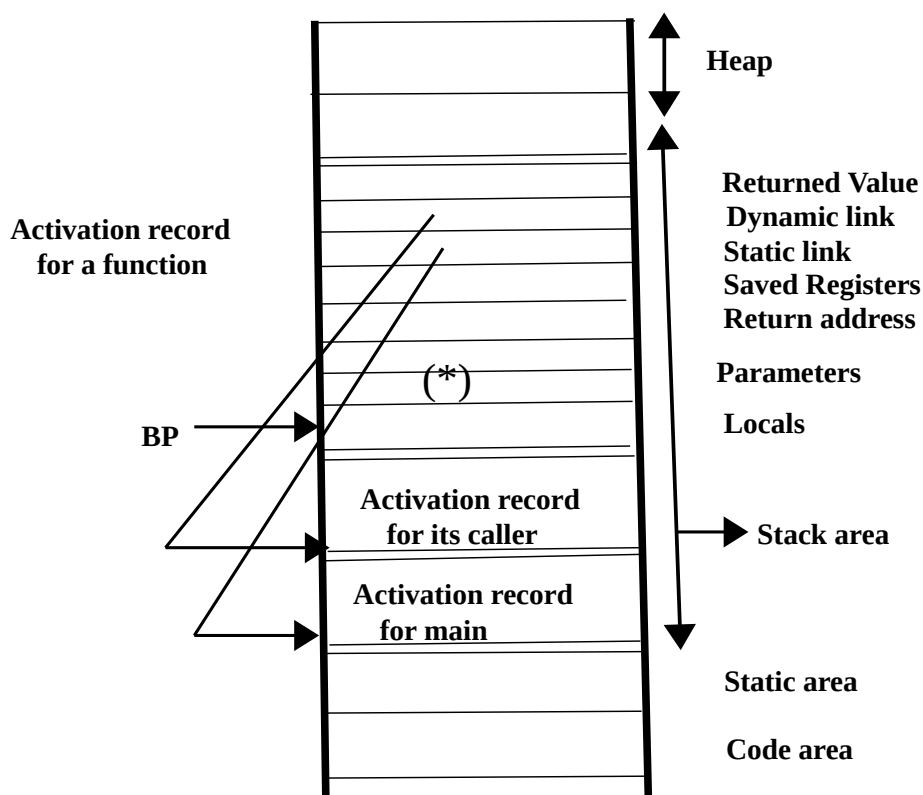
Reader : Draw the configuration at this point in time

10. The last instruction in the code of p() is “ret”.

ret	1. Restore instruction pointer to that of the caller function	1. popq %rip
------------	---	--------------

Reader : Draw the Final configuration here

Now that we have a good insight into the Run Time Environment issues that arise in the context of C language and x86 64bit architecture, the generic issues in RTE for most combinations of PL and an architecture are briefly outlined.



This is one of many possible organizations of an activation record.

ACCESSING INFORMATION IN ACTIVATION RECORD

Within an activation record, any information is referred to using an offset relative to a pointer (BP) pointing to the base of the activation record. Thus the address of the local variable y is BP (contents of register BP – offset for y) and the parameter x often has the address (contents of BP + offset) in a possible activation record layout for a function.

Some machines use a dedicated register for holding the base pointer. Actual contents of the activation record depends on the programming language and the machine architecture.

PARAMETER PASSING

The common methods to associate an actual parameter with a formal parameter are:

1. **Call by value** : The calling procedure evaluates the actual parameter and places the resulting value in the activation record of the called procedure.
2. **Call by reference** : The actual parameter is a variable. The calling procedure places the address of this variable in the activation record of the called procedure. A reference to the formal parameter, then, becomes an indirect reference through the address passed to the called procedure.

There are other methods of parameter passing that have been defined in different languages and also successfully implemented.

In this course, we shall consider the above two parameter passing methods only.

COMPILING PROCEDURE CALLS

During a procedure call, there are many possible ways of dividing the bookkeeping work between the caller and the callee.

One possible way is:

During a procedure call

Tasks performed by Caller

1. Makes space on the stack for a return value.
2. Puts the actual parameters on the stack.
3. Sets the static link. **[Not relevant for C/C++: May be ignored]**
4. Jumps to the called procedure. Return address is saved on the stack.

Tasks performed by Callee

1. Sets the dynamic link.
2. Sets the base of the new activation record.
3. Saves registers on the stack.
4. Makes space for local variables on the stack.

COMPILING PROCEDURE CALLS

Note that if we follow this sequence, the base pointer does not actually point to the base of the activation record, but somewhere in the middle. The actual parameters are now referenced using a negative offset, and the local variables with a positive offset.

During a return

Tasks performed by Callee

1. Restores registers.
2. Sets the base pointer to the activation record of the calling procedure.
3. Returns to the caller.

Tasks performed by Caller

1. Restores stackpointer to the location containing the returned value.

COMPILING PROCEDURE CALLS

X86-64 BIT ARCHITECTURAL SUPPORT FOR COMPILATION

During a procedure call : *Tasks performed by Caller*

1. Makes space on the stack for a return value / use a dedicated register

mov \$0, %eax

2. Passing actual arguments to callee

- First 6 arguments through dedicated registers :

movl -64(%rbp), %rcx # arg4

movl -68(%rbp), %rdx # arg3

movl -56(%rbp), %rsi # arg2

movq -16(%rbp), %rdi # arg1

- The remaining arguments are saved on the stack

movl -44(%rbp), %edi # arg9

pushq %rdi #

movl -48(%rbp), %edi # arg8

pushq %rdi #

movl -52(%rbp), %edi # arg7

pushq %rdi #

3. Sets the static link.

Not required for C / C++

4. Jumps to the called procedure. Return address is saved on the stack.

call f

Equivalent to **pushq %rip**

jmp \$ 0x0604420 OR **addl \$0x0200, %rip**

depending on whether address of f() is referred using an absolute address, such as **\$0x0604420** or a relative address, **\$0x0200**, with respect to %rip

During a procedure call : Tasks performed by Callee

1. Sets the dynamic link.

pushq %rbp // pushes the base pointer on the stack

2. Sets the base of the new activation record.

movq %rsp, %rbp //Sets the base pointer to the top of the stack

3. Saves registers on the stack.

If caller has passed arguments through dedicated registers, then these are saved by the callee on the stack.

movl %edi, -20(%rbp) # save arg1 on stack

movl %esi, -24(%rbp) # save arg2 on stack

movl %edx, -28(%rbp) # save arg3 on stack

movq %rcx, -40(%rbp) # save arg4 on stack

On some architectures, there are special instructions to save a group of registers on the stack.

4. Makes space for local variables on the stack.

subq \$48, %rsp

Makes disp amount of space on the top of the stack.

disp is usually negative because the stack grows towards decreasing # memory locations in x86.

During a return from a call : tasks performed by Callee

1. Restores registers. Not used here unless it is inevitable. The pre-assigned protocol of registers to caller and callee normally suffices without a need for restoring registers across a call.

2. Sets the base pointer to the activation record of the calling procedure.

Leave

This instruction is equivalent to

movl %rbp, %rsp

popq %rbp

3. Returns to the caller.

ret

This instruction restores the return address by resetting the old value of %rip saved on the stack at the time of call.

During a return from a call : tasks performed by Caller

1. Restores stack pointer to the location containing the returned value.

movq %rax, -20(%rbp) # return value is assigned to a name

Take Home Assignment :

For all the program fragments used in the examples, draw the AR for each of the functions with respect to the caller-callee conventions. Show the relative address of all the local variables and parameters on the stack (if applicable) and the passing of parameters from actual to formal using the registers dedicated for our architecture.

RUNTIME ENVIRONMENTS : A SUMMARY

BASIC CONCEPTS & ISSUES

To decide on the organization of data objects, so that their addresses can be resolved at compile time.

- The data objects (represented by variables) come into existence during the execution of the program.
- The generated code must refer to the data objects using their addresses, which must be resolved during compilation.
- The addresses of data objects depend on their organization in memory. This is largely decided by source language features.

We shall restrict this module to study of **Programming language environments** in which

- The sizes of objects and their relative positions in memory are known at compile time.
- Recursion is permitted.
- Data structures can be created at run time.

We call such a allocation as **stack allocation**. C, C++ and Pascal are among the languages that support such environments.

The organization of data is largely determined by answers to questions like the following.

1. **Does the language permit recursion?** There may be many incarnations of factorial active at the same time, each with its copy of local variables and parameters which must be allocated storage. Moreover, the number of such incarnations are only known at run-time.
2. **What are the parameter passing mechanisms?** Inside factorial, the access mechanisms for the formal parameters *r* and *x* have to be different. This is because *r* is **call-by-reference**, whereas *x* is **call-by-value**.
3. **How does a procedure refer to the non-local names?** The **rules of static scoping** decide that the *z* being referenced in factorial refers to the *z* in the main function. Therefore, apart from the local variables and formal parameters, factorial should be able to access this variable at run-time.

4. **Does the language permit creation of dynamic data structures?** Space must be created at run-time, each time new(y) is executed. What are the requirements of space allocation for locals, non-locals, parameters and dynamic data ?

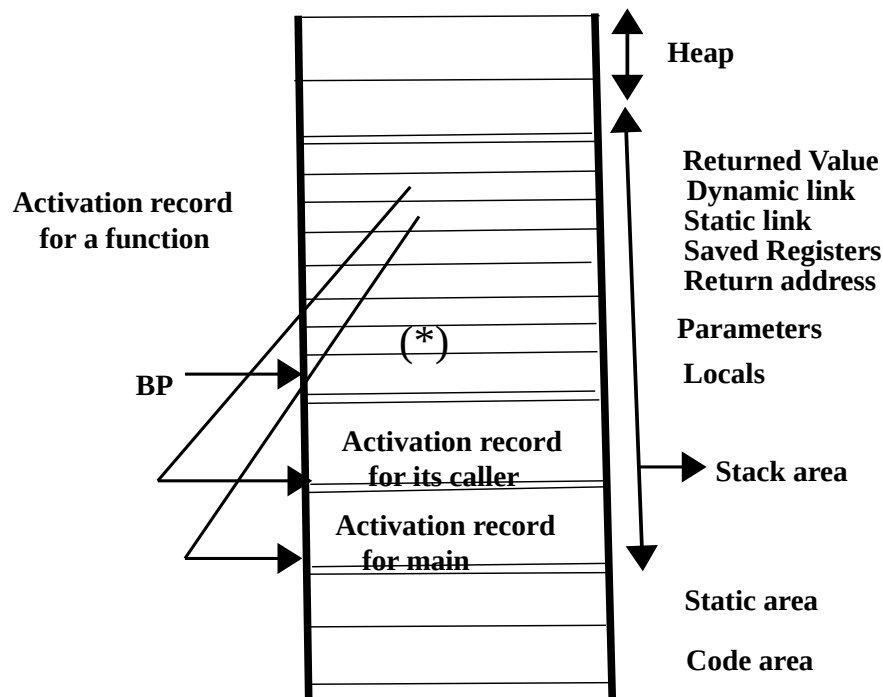
ACTIVATION RECORDS

An activation record contains the information required for a single activation of a procedure. Activation records are held in the static area for languages like Fortran and in the stack area for languages like C, C++, Pascal. Typically space must be provided in the activation record for the following.

1. **Local variables:** Part of the local environment of a procedure.
2. **Parameters:** Also part of the local environment.
3. **Return address:** To return to the appropriate control point of the calling procedure.
4. **Saved registers :** If the called procedure wants to use the registers used by the calling procedure, these have to be saved before and restored after the execution of the called procedure.
5. **Static link :** For accessing the non-local environment. The **static link** of a procedure points to the latest activation record of the immediately enclosing procedure. Static links, may be avoided by using a **display mechanism**. A static link is not required for Fortran IV like languages.
6. **Dynamic link :** Pointer to activation record of calling procedure.
7. **Returned value:** Used to store the result of a function call.

HANDLING DYNAMIC DATA STRUCTURES

A pointer variable p is allocated memory on the stack. On execution of , memory is allocated from the heap and the home location of the pointer variable (on the stack) is made to point to the allocated location in the heap. Any reference to p-> is an indirect reference to the heap location through the address on the stack



***** End of Document *****

Birla Institute of Technology, Mesra
Department of Computer Science & Engineering
CS 333 : Compiler Design
TUTORIAL 1 (Manual Analysis); Jan 10, 2024

Consider the following C code given to you for manual analysis.

```
#include <stdio.h>
const int PRIME = 211;
int hashpjw(char* s)    // function definition
{
    char* p;
    unsigned h = 0, g;
    for ( p = s; *p != '\0'; p = p + 1 ) // start of loop
    {
        h = ( h << 4 ) + ( *p );
        if ( g = h & 0xf0000000 ) { h = h ^ ( g << 24 ); h = h ^ g; }
    } // end of loop
    return h % PRIME;
}
int main() { }
```

P1. Identify manually the tokens and lexemes that are present in the following program. Give reasonable names for the tokens (such as keyword, identifier, etc.). Present your findings in a tabular form given below. You may use the table given in the Appendix to refer to operators and their properties of C/C++ language.

Token No	Token Type	Lexeme
1	keyword	int
2	identifier	hashpjw
..

P2. Use a tuple representation for a token, (token-type, lexeme). Show the stream of tokens generated after the entire code has been tokenized all white space (space, newline, tab) has been stripped off.

P3. Write regular expressions to specify the following lexemes in the given C code. Write a separate regular expression for each distinct token type.

- White-space
- const int for unsigned
- PRIME s p h g
- 211 0 4 0xf0000000
- = != << * ^ () { }
- , ; ' ' ,

P4. Write a regular expression for recognizing single line comments in C/C++.

APPENDIX : OPERATORS & THEIR ATTRIBUTES IN C / C++

Precedence	Associativity	Arity	Operator	Function of the operator
16	L	binary	[]	array index
15	R	unary	++, --	increment, decrement
15	R	unary	~	bitwise NOT
15	R	unary	!	logical NOT
15	R	unary	+, -	unary minus, plus
15	R	unary	*, &	dereference, address of
13	L	binary	*, /, %	multiplicative operators
12	L	binary	+, -	arithmetic operators
11	L	binary	<<, >>	bitwise shift
10	L	binary	<, <=, >, >=	relational operators
9	L	binary	==, !=	equality, inequality
8	L	binary	&	bitwise AND
7	L	binary	^	bitwise XOR
6	L	binary	 	bitwise OR
5	L	binary	&&	logical AND
4	L	binary	 	logical OR
3	L	ternary	?:	arithmetic if
2	R	binary	=, *=, /=, %=, +=, -=, <<=, >>=, &=, =, ^=	assignment operators

End of Tutorial Sheet 1

Birla Institute of Technology, Mesra
Department of Computer Science & Engineering
CS 333 : Compiler Design
TUTORIAL 2 (Lexical Analysis); Jan 22, 2024

Definitions, Algorithms and related information are given at the end of the sheet.

Skill set : Writing regular expressions for PL tokens; ability to manually design a dfa directly from regex; represent a dfa in the 4 array scheme.

P1. Find the errors in the following regular expression specification for floating point numbers, with the underlying alphabet {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, •(decimal point)}. Report errors, if any, by giving instances of a) valid floating point numbers that not generated by the regex, and b) invalid float values that are generated by the regex.

pattern-name	regex
float	{digit}*\. {digit}*{digit}+

If the regex has errors, then write the corrected version.

P2. Modify the regular expression of P1 so that floating point values of the following form are also captured over and above the float values covered by P1, {1.2e+3, 0.234e3, .23e-21, 256e-1,}.

P2. Write regular expression for recognizing single line comments in C/C++. The action required is to count the number of single line comments in a file when you write the specifications in lex.

P3. Consider the regular expressions for the three distinct tokens given below.

(aba)+	Token 1
(ab*a)	Token 2
a(a b)	Token 3

Construct a labeled syntax tree for a combination of any 2 of the three tokens given above. Use the symbol '#' as end marker of a token.

P4. For the syntax tree constructed for 2 tokens (chosen by you) in P3

(aba)+	Token 1
(a(b)*a)	Token 2
a(a b)	Token 3

- (a) Calculate nullable(), firstpos() and lastpos() information and attach to nodes in the syntax tree
- (b) Identify all internal nodes where followpos() needs to be constructed, and compute followpos() at each relevant node

P5. Using the sets computed in P4, construct the DFA using the direct algorithm.

- (a) Manually check the correctness of the DFA constructed.
- (b) In case there is a final state with more than one token associated with it, produce input string(s) that matches with all the associated tokens.

P6. This question concerns the four array representation scheme for a DFA. The function used. Nextstate(), used is reproduced below. Find the minimal and maximal number of 1-dimensional array references required to find the transition, nextstate[s, a], for a state given s on character c.

```
function nextstate (s, a)
{ if CHECK[BASE[s] + a] == s
  then NEXT[BASE[s] + a]
  else return (nextstate( DEFAULT[s], a))
}
```

P7. Construct manually the 4 array based representation for the following DFA in terms of Default, Base, Next and Check such that array sizes are as low as possible. The final states are {1, 2}.

States	a	b	c	d
0	5	4	-	-
1	1	-	-	1
2	-	-	6	5
3	-	-	-	2
4	5	4	-	1
5	2	3	4	4
6	5	4	-	-

P8. Examine with reason whether the DFA of P7 is minimal? If not then minimize it.

Take Home Assignment : Consider the following 2 regular expressions for identifier and number

letter	a b c d e f g h
digit	1 2 3 4 5
Identifier	letter (letter digit)*
number	digit(0 digit)*

You are required to manually go through all the steps starting from

- creation of syntax tree
- direct dfa construction using the 4 sets of information
- minimizing the dfa constructed
- laying out the minimized dfa in 4 array scheme

APPENDIX 1 : DEFINITIONS AND ALGORITHMS

The basic regular expression operators and the language denoted by them. Note that the tool “lex” may offer additional

Table 1 : Regular Expression Operators and Underlying Languages

Expression	Describes	Language	Example
ϵ	Empty string	$\{\epsilon\}$	
c	Any non-meta linguistic character c	$\{c\}$	a
r^*	Zero or more r 's	$L(r)^*$	a^*
r^+	One or more r 's	$L(r)^+$	a^+
$r1 \cdot r2$	$r1$ followed by $r2$	$L(r1) \cdot L(r2)$	$a \cdot b$ or ab
$r1 \mid r2$	$r1$ or $r2$	$L(r1) \cup L(r2)$	$a \mid b$
(r)	r	$L(r)$	$(a \mid b)$

Table 2 : Properties of Regular Expression Operators

	Regular expression operators with precedence values			
Operator	Parentheses ()	Closure * Reflexive closure +	Concatenation •	Alternate
Precedence	4	3	2	1
Associativity	Left	Right	Left	Left

Table 3 : Rules for computation of the four functions

node n	$nullable(n)$	$firstpos(n)$	$lastpos(n)$	$followpos(i)$
Leaf labeled ϵ	true	\emptyset	\emptyset	
Leaf labeled with $a \in \Sigma$ at position i	false	$\{i\}$	$\{i\}$	
$c_1 \mid c_2$	$nullable(c_1)$ or $nullable(c_2)$	$firstpos(c_1) \cup firstpos(c_2)$	$lastpos(c_1) \cup lastpos(c_2)$	
$c_1 \cdot c_2$	$nullable(c_1)$ and $nullable(c_2)$	if $nullable(c_1)$ then $firstpos(c_1) \cup firstpos(c_2)$ else $firstpos(c_1)$	if $nullable(c_2)$ then $lastpos(c_1) \cup lastpos(c_2)$ else $lastpos(c_2)$	if $i \in lastpos(c_1)$ then $followpos(i) += firstpos(c_2)$
c^*	true	$firstpos(c)$	$lastpos(c)$	if $i \in lastpos(c)$ then $followpos(i) += firstpos(c)$
c^+	$nullable(c)$	$firstpos(c)$	$lastpos(c)$	if $i \in lastpos(c)$ then $followpos(i) += firstpos(c)$

REGEX TO DIRECT DFA ALGORITHM

1. Construct the tree for $r\#$ for the given regular expression r .
2. Construct functions *nullable()*, *firstpos()*, *lastpos()* and *followpos()*
3. Let *firstpos(root)* be the start state. Push it on top of a stack.
While (stack not empty)
do begin
 pop the top state U off the stack; mark it;
 for each input symbol a do
 begin
 let $p_1, p_2, p_3, \dots p_k$ be the positions in U corresponding to symbol a ;
 Let $V = \text{followpos}(p_1) \cup \text{followpos}(p_2) \cup \dots \cup \text{followpos}(p_k)$;
 place V on stack if not marked and not already in stack;
 make a transition from U to V labeled a ;
 end
end
end
4. Final states are the states containing the positions corresponding to $\#$.

FOUR ARRAY REPRESENTATION :

Default[] and Base[] are the same size as the number of states. The size of the Next[] and Check[] arrays depend on the extent of exploitation of sparsity and common transitions among different states. The generic structure is included below.

Index	Default	Base		Index	Next	Check
0				0		
1				1		
2				2		
3				3		
4				4		
				5		
				6		

Table 4 : Four Array data structure for representing a DFA

The calculation of the next state information in a 4array scheme is outlined in the following function.

```
function nextstate (s, a)
{ if CHECK[BASE[s] + a] = s
  then NEXT[BASE[s] + a]
  else
    return (nextstate( DEFAULT[s], a))
}
```

A heuristic, which works well in practice to fill up the four arrays, is to find for a given state, the lowest BASE, so that the special entries of the state can be filled without conflicting with the existing entries.

APPENDIX 2

OPERATORS & THEIR ATTRIBUTES IN C / C++

Precedence	Associativity	Arity	Operator	Function of the operator
16	L	binary	[]	array index
15	R	unary	++, --	increment, decrement
15	R	unary	~	bitwise NOT
15	R	unary	!	logical NOT
15	R	unary	+, -	unary minus, plus
15	R	unary	*, &	dereference, address of
13	L	binary	*, /, %	multiplicative operators
12	L	binary	+, -	arithmetic operators
11	L	binary	<<, >>	bitwise shift
10	L	binary	<, <=, >, >=	relational operators
9	L	binary	==, !=	equality, inequality
8	L	binary	&	bitwise AND
7	L	binary	^	bitwise XOR
6	L	binary		bitwise OR
5	L	binary	&&	logical AND
4	L	binary		logical OR
3	L	ternary	?:	arithmetic if
2	R	binary	=, *=, /=, %= +=, -=, <=, >>=, &=, =, ^=	assignment operators

End of Tutorial Sheet 2

CS 333 : Compiler Design

TUTORIAL 3 (Grammars and Top down Parsing) : Released Monday February 05, 2024

The operators in C++ with their attributes given below are used in several problems on this sheet.

Precedence	Associativity	Arity	Operator	Function
16	L	binary	[]	array index
15	R	unary	++, --	increment, decrement
15	R	unary	~	bitwise NOT
15	R	unary	!	logical NOT
15	R	unary	+,	unary minus, plus
15	R	unary	*, &	dereference, address of
13	L	binary	*, /, %	multiplicative operators
12	L	binary	+, -	arithmetic operators
11	L	binary	<<, >>	bitwise shift
10	L	binary	<, <=, >, >=	relational operators
9	L	binary	==, !=	equality, inequality
8	L	binary	&	bitwise AND
7	L	binary	^	bitwise XOR
6	L	binary		bitwise OR
5	L	binary	&&	logical AND
4	L	binary		logical OR
3	L	ternary	?:	arithmetic if
2	R	binary	=, *=, /=, %= +=, -=, <=<= >>=, &=, =, ^=	assignment operators

Problems 1 through 7 are similar in nature. The more you practice these problems, your skills in grammar writing, applying grammar transformations, constructing FIRST() and FOLLOW() sets, etc. will get sharpened.

P1. (a) Construct a parse tree for the following expression :

a += a *= a /= 5 + a && 4 || 10 - a

(b) Given that a has the value 5 at start, use your parse tree to determine the value of **a** after evaluation of the expression. Show the value at each node of the parse tree. Write a program and verify that your parse tree AND/OR evaluation is correct.

(c) Write an unambiguous expression grammar for the following operators {=, +=, *=, %=, |, &&, binary +, binary -} and operands are tokens **num** and **id**.

(d) In case your grammar has left recursion, eliminate left recursion and also perform left factoring, if applicable, to produce an equivalent grammar without left recursion and without common left factors for all alternates of a rule.

(e) Construct FIRST and FOLLOW sets for your grammar and then construct LL(1) parsing table. Is the grammar LL(1)? Justify your answer.

(f) Parse the expression of part (a) using your parser and report the results.

P2. Consider the set of operators {=, +=, |, &&, ==, !=, <, <=, >, >=, binary +} and operand set = {**num**, **id**}.

(a) Create an expression involving all the operators and operands as relevant, on the same lines as the expression of P1 part (a).. Construct a parse tree for your expression.

(b) Assume initial values for the **id**'s used in the expression and use your parse tree to evaluate the expression. Show the value at each node of the parse tree. Write a program and verify that your parse tree AND/OR evaluation is correct.

(c) Write an unambiguous expression grammar for the operator ad operand set of part (a).

(d) In case your grammar has left recursion, eliminate left recursion and also perform left factoring, if applicable, to produce an equivalent grammar without left recursion and without common left factors for all alternates of a rule.

(e) Construct FIRST and FOLLOW sets for your grammar and then construct LL(1) parsing table. Is the grammar LL(1) ? Justify your answer.

(f) Parse the expression of part (a) using your parser and report the results.

P3. Consider the set of operators {=, |, &&, ==, !=, binary +, binary -, *, /, %, unary -} and operand set = {**num**, **id**}. **The * is the multiplicative operator.**

(a) Create an expression involving all the operators and operands as relevant, on the same lines as the expression of P1 part (a).. Construct a parse tree for your expression.

(b) Assume initial values for the **id**'s used in the expression and use your parse tree to evaluate the expression. Show the value at each node of the parse tree. Write a program and verify that your parse tree AND/OR evaluation is correct.

(c) Write an unambiguous expression grammar for the operator ad operand set of part (a).

(d) In case your grammar has left recursion, eliminate left recursion and also perform left factoring, if applicable, to produce an equivalent grammar without left recursion and without common left factors for all alternates of a rule.

(e) Construct FIRST and FOLLOW sets for your grammar and then construct LL(1) parsing table. Is the grammar LL(1) ?

(f) Parse the expression of part (a) using your parser and report the results.

P4. Consider the set of operators {=, |, &&, ==, !=, <<, >>, bitwise and &, bitwise xor ^, bitwise or |, bitwise not ~, logical not !} and operand set = {**num**, **id**}.

(a) Create an expression involving all the operators and operands as relevant, on the same lines as the expression of P1 part (a).. Construct a parse tree for your expression.

(b) Assume initial values for the **id**'s used in the expression and use your parse tree to evaluate the expression. Show the value at each node of the parse tree. Write a program and verify that your parse tree AND/OR evaluation is correct.

(c) Write an unambiguous expression grammar for the operator and operand set of part (a).

(d) In case your grammar has left recursion, eliminate left recursion and also perform left factoring, if applicable, to produce an equivalent grammar without left recursion and without common left factors for all alternates of a rule.

(e) Construct FIRST and FOLLOW sets for your grammar and then construct LL(1) parsing table. Is the grammar LL(1) ?

(f) Parse the expression of part (a) using your parser and report the results.

P5. Write an unambiguous expression grammar involving all the operators of precedence levels 4 to 6, 12 and 13. Eliminate left recursion and left factoring from your grammar, if possible.

P6. Consider the set of operators {=, ++, --, dereference *, address of &, ==, !=, binary +, binary -, multiplicative *, multiplicative /} and operand set = {**num**, **id**}.

(a) Create an expression involving all the operators and operands as relevant, on the same lines as the expression of P1 part (a).. Construct a parse tree for your expression.

(b) Assume initial values for the **id**'s used in the expression and use your parse tree to evaluate the expression. Show the value at each node of the parse tree. Write a program and verify that your parse tree AND/OR evaluation is correct.

(c) Write an unambiguous expression grammar for the operator and operand set of part (a).

(d) In case your grammar has left recursion, eliminate left recursion and also perform left factoring, if applicable, to produce an equivalent grammar without left recursion and without common left factors for all alternates of a rule.

(e) Construct FIRST and FOLLOW sets for your grammar and then construct LL(1) parsing table. Is the grammar LL(1) ?

(f) Parse the expression of part (a) using your parser and report the results.

P7. Consider the set of operators {=, ++, --, dereference *, address of &, ==, !=, binary +, multiplicative *, array reference [] and ternary ?: } and operand set = {num, id }.

(a) Create an expression involving all the operators and operands as relevant, on the same lines as the expression of P1 part (a).. Construct a parse tree for your expression.

(b) Assume initial values for the id's used in the expression and use your parse tree to evaluate the expression. Show the value at each node of the parse tree. Write a program and verify that your parse tree AND/OR evaluation is correct.

(c) Write an unambiguous expression grammar for the operator and operand set of part (a).

(d) In case your grammar has left recursion, eliminate left recursion and also perform left factoring, if applicable, to produce an equivalent grammar without left recursion and without common left factors for all alternates of a rule.

(e) Construct FIRST and FOLLOW sets for your grammar and then construct LL(1) parsing table. Is the grammar LL(1) ?

(f) Parse the expression of part (a) using your parser and report the results.

P8. Examine the following grammar for if-then-else ambiguity; assume the symbols if, then, else, expr to be terminals for this problem. In case the given grammar does not serve the required purpose, write an unambiguous grammar that works.

$$\begin{aligned} stmt &\rightarrow \text{if expr then } stmt \mid matched_statement \\ matched_statement &\rightarrow \text{if expr then } matched_statement \text{ else } stmt \end{aligned}$$

P9. Examine the following grammar for if-then-else. Assume that the symbols { if, then, else, expr, other } to be terminals for this problem.

$$\begin{aligned} S &\rightarrow M \mid U \\ M &\rightarrow \text{if expr then } M \text{ else } M \mid \text{other} \\ U &\rightarrow \text{if expr then } S \mid \text{if expr then } U \text{ else } M \end{aligned}$$

Consider the string **if expr then if expr then other else if expr then other else other**

Answer the following questions

(a) Find as many distinct leftmost derivations of the string as possible.

- (b) Show the parse tree corresponding to each derivation of part (a) above
 (c) Comment on ambiguity of this grammar ?

P10. Given the following specification of a language through regular expressions

$variable \rightarrow (simple_or_array \cdot)^* | simple_or_array$
 $simple_or_array \rightarrow id [elist] ?$
 $elist \rightarrow (e,)^* e$

where e is a terminal.

- (a) Write a LL(1) grammar specifying the same language.
 (b) Construct a LL(1) parsing table for the same grammar.

P11. Consider the following context free grammar.

$S \rightarrow X$
 $X \rightarrow Ma | bMc | dc | bda$
 $M \rightarrow d$

- (a) Determine whether the grammar as given is LL(1). Report all conflicts if the grammar is not LL(1).
 (b) In case answer to (a) is no, Is it possible to rewrite this grammar to an equivalent LL(1) grammar ?

Write LL(1) parsers for the above grammar.

P12. Write a grammar for declarations in language C, that can generate one or more variables declared in the same statement and one or more declaration statements. The built-in types to be used are int and float and scalars only. Sample declarations are given below.

int a, b, c;
 float x;
 int * p, q;

P13. Consider the following context free grammar. You have to determine whether the grammar is LL(k), where k = 1 or 2. Construct and report the FIRST() and FOLLOW() sets. In case you find that the grammar is not LL(k), show any multiple-entry of the corresponding LL(k) table without constructing the entire table. In case you find that the grammar is indeed LL(k), argue it with respect to FIRST() AND FOLLOW() information.

1) $S \rightarrow G id$ 2) $G \rightarrow P G'$ 3,4) $G' \rightarrow G | \epsilon$ 5) $P \rightarrow id : R$
 6) $R \rightarrow id R'$ 7,8) $R' \rightarrow R | ;$

P14. Identify which of the grammars given below is LL(1) and why ?

- | | | |
|------------------------------|------------------------------|-------------------------------------|
| (a) $S \rightarrow A B c$ | (b) $S \rightarrow A B B A$ | (c) $E \rightarrow - E (E) V R$ |
| $A \rightarrow a \epsilon$ | $A \rightarrow a \epsilon$ | $V \rightarrow id T$ |
| $B \rightarrow b \epsilon$ | $B \rightarrow b \epsilon$ | $R \rightarrow - E \epsilon$ |
| | | $T \rightarrow (E) \epsilon$ |

Write LL(1) parsers for each of the above grammars.

P15. Write a grammar for array declarations, which beside the declarations for scalar variables, also admits multi dimensional arrays of C. Sample declarations that are to be generated are :

```
int a, b[20], c;  
float x, y[5][2];  
int * p[2][2][5], q;
```

******* End of Tutorial Sheet 3 *******

CS333 : Compiler Design

TUTORIAL 4 (Bottom Up Parsing : SLR Parsing) Posted : February 12, 2024

P1. We have seen that left recursive rules in a context free grammar are not suitable for top down parsing. Does bottom up parsing have an analogous problem ? Examine any bottom up SLR(1) parser with respect to recursive rules, left recursive or right recursive, and justify your observations.

P2. The stack contents and the input symbol at some point during parsing by a shift reduce parser is shown below with the nonterminal C on top of the stack. The terminals on the stack are {a, b, c, d} and the nonterminals are {A, B, D, C}

STACK : a A b B c D d C

Identify all the potential handles in the stack.

P3. Explain why the blank entries in the Goto part of a SLR (1) parsing table are not error entries in the usual sense of similar entries in the Action part.

P4. A LR(0) grammar is one for which the corresponding LR(0) parsing table has no conflicts; a reduce action in a state of an LR(0) parser is performed on all terminals T. Consider the following context free grammar.

$$\begin{aligned} S &\rightarrow X \\ X &\rightarrow Ma \mid bMc \mid dc \mid bda \\ M &\rightarrow d \end{aligned}$$

(a) Determine whether the grammar as given is LR(0). Report all the conflicts if your answer is in the negative.

(b) Recall that in SLR(1), a reduce action, $A \rightarrow \alpha$, is performed for all terminals in FOLLOW(A). Determine whether the grammar as given is SLR(1). Report all the conflicts if your answer is in the negative.

P5 (a) Identify which of the grammars, as given below, is LR(0) and/or SLR(1) and why ?

(i) $S \rightarrow A B c$	(ii) $E \rightarrow -E \mid (E) \mid V R$
$A \rightarrow a \mid \epsilon$	$V \rightarrow id T$
$B \rightarrow b \mid \epsilon$	$R \rightarrow -E \mid \epsilon$
	$T \rightarrow (E) \mid \epsilon$

(b) If the grammars as given are not SLR(1), can they be transformed to an equivalent SLR(1) ?

P6. We would like to process program fragments, as given below, for the purpose of parsing.

```
int a [10] [20];
int b [10] [20];
int i, j;
i = 10;
j = 16;
b[i+1] [ j+2] = a [i-1] [j+1] + a [i ] [j - 2] + 25;
```

Write a CFG so that such code fragment can be generated. Your grammar should be able to (i) generate declarations of scalars and arrays as denoted by the first 3 lines above, and (ii) also generate assignments involving arrays and scalars, as given in the last 3 lines of the sample code above. Assume that an array is stored in row-major representation , if such information is necessary.

(b) Determine whether the grammar written by you in part (a), will admit a SLR(1) parser by constructing the automaton or otherwise.

(c) Construct the parse tree for the program fragment of part (a) using your grammar and a SLR(1) parser.

P7. Construct a grammar, if possible, such that it

(a) is SLR(1) but not LL(1)

(b) is LL(1) but not SLR(1)

***** End of Tutorial Sheet 4 *****

CS333 : Compiler Design

TUTORIAL 5 : LR Parsing

P1. We have seen that left recursive rules in a context free grammar are not suitable for top down parsing. Does bottom up parsing have an analogous problem ? Examine any bottom up parser from the LR family with respect to recursive rules. Find out what difference exists, if any, when for the same language, first left recursive and then a right recursive grammar is used with your chosen bottom up parser.

P2. The stack contents and the input symbol at some point during parsing by a shift reduce parser is shown below with the nonterminal C on top of the stack. The terminals on the stack are {a, b, c, d} and the nonterminals are {A, B, D, C}

STACK : a A b B c D d C

Which among the following statements, S1 to S4, are true and why ?

S1 : Handles are { C, d C, D d C }

S2 : Handles are {D, D d, D d c }

S3 : Strings {A b B, c D d C} are not handles

S4 : The longest handle is of length 8.

P3. Examine the blank entries in the goto part of a LR parsing table (SLR(1), LALR(1) or LR(1)) are error entries in the sense that the blank entries are error in the Action part.

P4. Consider the following context free grammar.

$S \rightarrow X$
 $X \rightarrow Ma \mid bMc \mid dc \mid bda$
 $M \rightarrow d$

(a) Determine whether the grammar as given is LR(0). A LR(0) grammar is one for which the corresponding LR(0) parsing table has no conflicts; a reduce action in a state of an LR(0) parser is performed on all terminals. Report all the conflicts if your answer is in the negative. T. Recall that in SLR(1), a reduce action, $A \rightarrow \alpha$, is performed for all terminals in FOLLOW(A).

(b) Determine whether the grammar as given is SLR(1). Report all the conflicts if your answer is in the negative.

(c) Examine whether the grammar is LR(1) or LALR(1).

P5 (a) Identify which of the grammars given below, is LR(0) and/or SLR(1) and/or LALR(1) / LR(1) and why ?

(i) $S \rightarrow ABc$
 $A \rightarrow a \mid \epsilon$
 $B \rightarrow b \mid \epsilon$

(ii) $S \rightarrow ABBA$
 $A \rightarrow a \mid \epsilon$
 $B \rightarrow b \mid \epsilon$

(iii) $E \rightarrow -E \mid (E) \mid VR$
 $V \rightarrow id \mid T$
 $R \rightarrow -E \mid \epsilon$
 $T \rightarrow (E) \mid \epsilon$

(b) Can these grammars be transformed to an equivalent LR(0) and/or SLR(1) ?

(c) Can these grammars be transformed to an equivalent LR(1) and LALR(1) ?

P6. Consider the grammar given below for declaration processing.

P → D
D → **id** : T
D → **procedure id** ; D ; B
T → **integer** | **real**
B → **body**

(a) The grammar written above was supposed to parse program fragments of the form given below. Argue whether grammar given will serve its intended purpose, in case your answer is no, rewrite the grammar to achieve the desired objective.

```
procedure p1;  
  a : real;  
  procedure p2;  
    b : integer;  
    procedure p3;  
      b : real;  
      body; // of p3  
    procedure p4;  
      a : integer;  
      body; // of p4  
    body; // of p2  
  body; // of p1
```

(b) Determine whether the grammar as given, or the one rewritten by you in part (a), will admit a SLR(1) parser by constructing the automaton or otherwise.

(c) Construct the parse tree for the program fragment of part (a) using your grammar and a SLR(1) parser.

(d) Construct LR(1) and LALR(1) parsing table for the grammar written by you in part (b). Construct the parse tree for the program fragment of part (a) using these tables.

P7. We would like to process program fragments, as given below only for the purpose of parsing.

```
integer a [10, 20];  
integer b [10, 20];  
integer i, j;  
i = 10;  
j = 16;  
b[i+1, j+2] = a [i-1, j+1] + a [i, j - 2] + 25;
```


(a) Make appropriate changes to the grammar given below for array references, if so required, so that code fragment, as given above, can be generated along with parsing. Your grammar should be able to (i) generate declarations of scalars and arrays as denoted by the first 3 lines above, and (ii) also generate assignments involving arrays and scalars, as given in the last 3 lines of the sample code above. Assume that an array is stored in row-major representation, if such information is necessary.

$$\begin{aligned} S &\rightarrow L = E \\ E &\rightarrow L \\ L &\rightarrow \text{elist}] \mid \mathbf{id} \mid \mathbf{num} \\ \text{elist} &\rightarrow \text{elist} , E \mid \mathbf{id} [E \end{aligned}$$

(b) Determine whether the grammar as given, or the one rewritten by you in part (a), will admit a SLR(1) / LALR(1) / LR(1) parser by constructing the automaton or otherwise. We need the smallest parser of the family that works for the grammar.

(c) Construct the parse tree for the program fragment of part (a) using your grammar and the parser.

(d) Construct LR(1) and LALR(1) parsing tables for the grammar written by you in part (b). Construct an input that is generated using all the rules and show its parse tree using the constructed tables.

P8. For the grammar shown below, answer the following questions.

$$\begin{aligned} S &\rightarrow \text{id} [E] := E \\ E &\rightarrow E + T \mid E \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

Construct any one complete automaton from the 3 parsers, and show the kernel items of each state. Identify all the states of this automation with a self loop and write the set of all viable prefixes for this state. Justify the validity of all the LR(0) items contained in such states, and report your findings.

P9. Consider the following context free grammar which can generate certain assignment statements involving array references and arithmetic operators assuming the usual properties of operators.

$$\begin{aligned} S &\rightarrow L := E & E &\rightarrow L \mid E + T \\ T &\rightarrow T * F \mid F & L &\rightarrow \text{elist}] \mid \mathbf{id} \\ \text{elist} &\rightarrow \text{elist} , E \mid \mathbf{id} [E & F &\rightarrow \mathbf{id} \end{aligned}$$

(a) Construct a SLR(1) parser for the grammar given above, rewriting the same, if required, without changing the underlying language.

(b) Construct LR(1) and LALR(1) parsing tables for the grammar of part (a).

P10. Consider the grammar given below for generating program fragments with control flow constructs. The term **relop** stands for relational operators {<, <=, >, >=, ==, !=}.

$S \rightarrow \text{if } B \text{ then } S \mid \text{if } B \text{ then } S \text{ else } S \mid \text{while } B \text{ do } S \mid \text{begin } SList \text{ end} \mid L := E$

$SList \rightarrow SList ; S \mid S$

$B \rightarrow B \text{ or } B \mid B \text{ and } B \mid \text{not } B \mid (B) \mid \text{true} \mid \text{false} \mid E \text{ relop } E \mid E$

$E \rightarrow E + E \mid \text{id}$

(a) The grammar written above was supposed to parse program fragments of the form given below. Argue whether grammar given will serve its intended purpose, in case your answer is no, rewrite the grammar to achieve the desired objective.

```

if ( a < b) then
  begin
    a := b + c;
    while ( a <= b) do begin a := c + d end;
    a := b
  end
else begin c := c + d end ;

```

(b) Determine whether your grammar will lead to a conflict free SLR(1) Parsing table. If the answer is in the negative, report the conflicts.

(c) Construct the parse tree for the program fragment of part (a) using your grammar and a SLR(1) parser.

(d) Construct LR(1) and LALR(1) parsing table for the grammar written by you in part (b). Construct the parse tree for the program fragment of part (a) using these tables.

P11. Construct a grammar

(a) that is SLR(1) but not LL(1)

(b) that is LL(1) but not SLR(1)

(c) that is LALR(1) but not SLR(1)

(d) that is LR(1) but not LALR(1)

P12. Identify the following grammars as LL(1), LR(0), SLR(1), LALR(1) or LR(1)

(a) $S \rightarrow d M N c \mid N \mid L$
 $N \rightarrow d b \mid c d$
 $L \rightarrow d M d M$
 $M \rightarrow b$

(b) $S \rightarrow d B \mid A d \mid B c$
 $A \rightarrow C c$
 $B \rightarrow a$
 $C \rightarrow d a$

P13. Consider the SLR(1) automaton for the pointer grammar G, given in the notes

0: $S' \rightarrow S$

1,2: $S \rightarrow L = R \mid R$

3,4: $L \rightarrow * R \mid id$

5: $R \rightarrow L$

- (a) Write 5 distinct strings from $(N \cup T)^*$ that are not viable prefixes for G
- (b) Consider a state, s , of this automaton that belong to a loop. Write all distinct viable prefixes for the state s .
- (c) Explain why all the LR(0) items in state s are valid for the viable prefixes associated with this state.

P14. Do P13 for canonical LR(1) and LALR(1) automaton.

******* End of Tutorial Sheet *******

CS 333 : Compiler Design
TUTORIAL : Runtime Environments : April 2024

The generic forms discussed in the lecture for calling convention and activation record structure, are given below for reference and are to be used for solving these problems. Activation record (AR) may contain the information required for a single activation of a procedure. However it is possible

- | | | |
|--------------------|----------------|-------------------|
| 1. Local variables | 2. Parameters | 3. Return address |
| 4. Saved registers | 5. Static link | 6. Dynamic link |
| | | 7. Return value |

Tasks performed by Caller (Caller prologue protocol : code inserted by compiler just before a call in the caller's body)	Tasks performed by Callee (Callee prologue protocol : code inserted by compiler just before the first executable statement in callee's body)
A1. Make space on the stack for a return value. A2. Pass actual parameters through stack or registers. A3. Sets the static link. A4. Save return address in caller's code on stack. A5. Jump to the called function	C1. Sets the dynamic link. C2. Sets base of new activation record. C3. Saves registers on the stack. C4. Makes space for locals on stack.
Tasks performed by Caller (Caller epilogue protocol : code inserted by compiler just after a call in the caller's body)	Tasks performed by Callee (Callee epilogue protocol : code inserted by compiler in the callee's body just before it returns control back to its caller)
B1. Picks return value from stack	D1. Restores registers. D2. Sets base pointer to AR of caller function. D3. Restores Stack pointer to location containing returned value. D4. Retrieves saved return address in caller's code and Returns back to caller

Use the brief description of X86-64 bit assembly given in the lecture notes.

Consider the C program spread out in 2 columns as given below for the two problems that follow.

<pre>#include <stdio.h> void f(int x, const int k) { x = x* k; if (x == 0) x = x + k; else x = x/2; printf(" x = %d k = %d \n", x, k); }</pre>	<pre>int main() { int a = 10, b = 2; f(a+a*b, b); return 0; }</pre>
--	---

P1. Examine the function f() with respect to the source and its annotated assembly given below in 2 columns.

- (a)** You have to identify all the callee prologue actions, C1 to C4, that are generally performed by a callee and are present in the given assembly code. You should be able to justify your answer.
- (b)** Identify all the callee epilogue actions, D1 to D4, that are generally performed by a callee and are present in the given assembly code with brief justifications.
- (c)** Examine the source code for all names used in its body, {x, k} and how they are referenced in the assembly code. Justify the relevant assembly instructions around the references.
- (d)** f() is caller for the printf() function. Examine the assembly code that relates to f() as a caller function, and identify the caller prologue and epilogue actions, {A1, ..., A5, B1} present in it.

```
void f( int x, const int k)
{
```

```
    x = x* k;
```

```
    if ( x == 0) x = x + k;
```

```
    else
```

```
        x = x/2;
```

```
    printf(" x = %d k = %d \n", x, k);
```

```
}
```

```
.LC0:
```

```
.string  " x = %d k = %d \n"
```

```
.text
```

```
.globl  f
```

```
.type   f, @function
```

```
f:
```

```
    pushq  %rbp  #
```

```
    movq   %rsp, %rbp  #,
```

```
    subq   $16, %rsp  #,
```

```
    movl   %edi, -4(%rbp) # x, x
```

```
    movl   %esi, -8(%rbp) # k, k
```

```
    movl   -4(%rbp), %eax # x, tmp88
```

```
    imull  -8(%rbp), %eax # k, tmp87
```

```
    movl   %eax, -4(%rbp) # tmp87, x
```

```
    cmpl   $0, -4(%rbp)  #, x
```

```
    jne    .L2  #,
```

```
    movl   -8(%rbp), %eax # k, tmp89
```

```
    addl   %eax, -4(%rbp) # tmp89, x
```

```
    jmp    .L3  #
```

```
.L2:
```

```
    movl   -4(%rbp), %eax # x, tmp91
```

```
    movl   %eax, %edx  # tmp91, tmp92
```

```
    shrl   $31, %edx  #, tmp92
```

```
    addl   %edx, %eax  # tmp92, tmp93
```

```
    sarl   %eax  # tmp94
```

```
    movl   %eax, -4(%rbp) # tmp94, x
```

```
.L3:
```

```
    movl   -8(%rbp), %edx # k, tmp95
```

```
    movl   -4(%rbp), %eax # x, tmp96
```

```
    movl   %eax, %esi  # tmp96,
```

```
    movl   $.LC0, %edi  #,
```

```
    movl   $0, %eax  #,
```

```
    call   printf  #
```

```
    nop
```

```
    leave
```

```
    ret
```

P2. Examine the function main() with respect to the source and its annotated assembly given below in 2 columns.

(a) You have to identify all the callee prologue and epilogue actions, C1 to C4 and D1 to D4, that are performed by main() as a callee in the assembly code given below with justifications.

(b) Identify all the caller prologue and epilogue actions , A1 to A4, B1 performed by main() as a caller of f() in its body with brief justifications.

(c) Examine the source code for all names used in its body, {a, b} and show how they are referenced in the assembly code. Justify the relevant assembly instructions around the references.

<pre> int main() { int a = 10, b = 2; f(a+a*b, b); return 0; } </pre>	<pre> .globl main .type main, @function main: pushq %rbp # movq %rsp, %rbp #, subq \$16, %rsp #, movl \$10, -8(%rbp) #, a movl \$2, -4(%rbp) #, b movl -4(%rbp), %eax # b, tmp91 addl \$1, %eax #, D.2302 imull -8(%rbp), %eax # a, D.2302 movl -4(%rbp), %edx # b, tmp92 movl %edx, %esi # tmp92, movl %eax, %edi # D.2302, call f # movl \$0, %eax #, D.2302 leave ret </pre>
---	---

Examine the source C program given below and answer the two problems that follow.

<pre> int f(int x1, int x2, int x3, int*p1, int x4, int x5, int x6, int*p2) { int a, b, c[100]; a = *p1 + x1 + x2 + x3 ; b = *p2 + x4 + x5 + x6; c[0]= 1; c[99] = 5; return a+b; } </pre>	<pre> int main() { int i1 = 1, i2 = 2, i3 = 3, i4 = 4, i5 = 5, i6 = 6, res; int *q1 = &i1, *q2 = &i2; res = f(i1, i2, i3, q1, i4, i5, i6, &i2); return 6; } </pre>
---	--

P3. Examine the function f() with respect to the source and its annotated assembly given below in 2 columns.

- (a) You have to identify all the callee prologue and epilogue actions, C1 to C4 and D1 to D4, that are performed by f() as evidenced in the given assembly code. Justify your answer briefly.
- (b) Examine the source code for all names used in the body of f() and explain why they are referenced in the assembly code in the manner that is done by gcc. Justify the relevant assembly instructions around the references.
- (c) Identify in the assembly code of f(), the instructions that relate to parameter passing mechanisms, correlate with the corresponding source code. Justify the appropriateness of the code generated by gcc for this purpose.

<pre> int f(int x1, int x2, int x3, int*p1, int x4, int x5, int x6, int*p2) { int a, b, c[100]; a = *p1 + x1 + x2 + x3 ; b = *p2 + x4 + x5 + x6; c[0]= 1; c[99] = 5; return a+b; } </pre>	<pre> movl %edi, -436(%rbp) # x1, x1 movl %esi, -440(%rbp) # x2, x2 movl %edx, -444(%rbp) # x3, x3 movq %rcx, -456(%rbp) # p1, p1 movl %r8d, -448(%rbp) # x4, x4 movl %r9d, -460(%rbp) # x5, x5 movq 24(%rbp), %rax # p2, tmp95 movq %rax, -472(%rbp) # tmp95, p2 movq %fs:40, %rax #, tmp113 movq %rax, -8(%rbp) # tmp113, D.2325 xorl %eax, %eax # tmp113 movq -456(%rbp), %rax # p1, tmp96 movl (%rax), %edx # *p1_2(D), D.2324 movl -436(%rbp), %eax # x1, tmp97 addl %eax, %edx # tmp97, D.2324 movl -440(%rbp), %eax # x2, tmp98 </pre>
---	--

<pre> .globl f .type f, @function f: pushq %rbp # movq %rsp, %rbp #, subq \$480, %rsp #, </pre>	<pre> movl %edi, -436(%rbp) # x1, x1 movl %esi, -440(%rbp) # x2, x2 movl %edx, -444(%rbp) # x3, x3 movq %rcx, -456(%rbp) # p1, p1 movl %r8d, -448(%rbp) # x4, x4 movl %r9d, -460(%rbp) # x5, x5 movq 24(%rbp), %rax # p2, tmp95 movq %rax, -472(%rbp) # tmp95, p2 movq %fs:40, %rax #, tmp113 movq %rax, -8(%rbp) # tmp113, D.2325 xorl %eax, %eax # tmp113 movq -456(%rbp), %rax # p1, tmp96 movl (%rax), %edx # *p1_2(D), D.2324 movl -436(%rbp), %eax # x1, tmp97 addl %eax, %edx # tmp97, D.2324 movl -440(%rbp), %eax # x2, tmp98 </pre>
--	--

```

addl    %eax,%edx    # tmp98, D.2324
movl    -444(%rbp), %eax    # x3, tmp102
addl    %edx,%eax    # D.2324, tmp101
movl    %eax, -424(%rbp)    # tmp101, a
movq    -472(%rbp), %rax    # p2, tmp103
movl    (%rax), %edx    # *p2_10(D), D.2324
movl    -448(%rbp), %eax    # x4, tmp104
addl    %eax,%edx    # tmp104, D.2324
movl    -460(%rbp), %eax    # x5, tmp105
addl    %eax,%edx    # tmp105, D.2324
movl    16(%rbp), %eax    # x6, tmp109
addl    %edx,%eax    # D.2324, tmp108

```

```

movl    %eax, -420(%rbp)    # tmp108, b
movl    $1, -416(%rbp)    #, c
movl    $5, -20(%rbp)    #, c
movl    -424(%rbp), %edx    # a, tmp110
movl    -420(%rbp), %eax    # b, tmp111
addl    %edx,%eax    # tmp110, D.2324
movq    -8(%rbp), %rcx    # D.2325, tmp114
xorq    %fs:40, %rcx    #, tmp114
leave
ret

```

P4. Examine the function main() with respect to the source and its annotated assembly given below in 2 columns.

(a) You have to identify all the callee prologue and epilogue actions, C1 to C4 and D1 to D4, that are performed by main() as a callee in the assembly code given below with justifications.

(b) Identify all the caller prologue and epilogue actions, A1 to A4, B1 performed by main() as a caller of f() in its body with brief justifications.

(c) Examine the source code for all names used in its body, and show how they are referenced in the assembly code. Justify the relevant assembly instructions around the references.

(d) Identify in the assembly code of main(), the instructions that relate to parameter passing mechanisms, correlate with the corresponding source code. Justify the appropriateness of the code generated by gcc for this purpose.

```

int main()
{
    int i1 = 1, i2 = 2, i3 = 3, i4 = 4, i5 = 5, i6 = 6, res;
    int *q1 = &i1, *q2 = &i2;
    res = f(i1, i2, i3, q1, i4, i5, i6, &i2);
    return 6;
}

```

```

        .globl  main
        .type   main, @function

main:
    pushq    %rbp    #
    movq     %rsp, %rbp    #,
    subq     $64, %rsp    #,
    movq     %fs:40, %rax    #, tmp101
    movq     %rax, -8(%rbp) # tmp101, D.2329
    xorl     %eax, %eax    # tmp101
    movl     $1, -52(%rbp) #, i1
    movl     $2, -48(%rbp) #, i2
    movl     $3, -44(%rbp) #, i3
    movl     $4, -40(%rbp) #, i4
    movl     $5, -36(%rbp) #, i5
    movl     $6, -32(%rbp) #, i6

```

```

    leaq     -52(%rbp), %rax    #, tmp91
    movq     %rax, -24(%rbp)    # tmp91, q1
    leaq     -48(%rbp), %rax    #, tmp92
    movq     %rax, -16(%rbp)    # tmp92, q2
    movl     -48(%rbp), %esi    # i2, D.2328
    movl     -52(%rbp), %eax    # i1, D.2328
    movl     -36(%rbp), %r9d    # i5, tmp93
    movl     -40(%rbp), %r8d    # i4, tmp94
    movq     -24(%rbp), %rcx    # q1, tmp95
    movl     -44(%rbp), %edx    # i3, tmp96
    leaq     -48(%rbp), %rdi    #, tmp97
    pushq    %rdi    # tmp97
    movl     -32(%rbp), %edi    # i6, tmp98
    pushq    %rdi    # tmp98
    movl     %eax, %edi    # D.2328,
    call     f    #
    addq     $16, %rsp    #,
    movl     %eax, -28(%rbp)    # tmp99, res
    movl     $6, %eax    #, D.2328
    movq     -8(%rbp), %rdx    # D.2329, tmp102
    xorq     %fs:40, %rdx    #, tmp102
    leave
    ret

```

End of Tutorial Problems