

CS419 Compilers Construction

A Simple One-Pass Compiler [Chapter 2]

Lecture 8

Edited by Dr. Ismail Hababeh

*Adapted from slides by Dr. Mohammad Daoud
German Jordanian University*

Originally from slides by Dr. Robert A. Engelen

Symbol Table Initialization - Example

The global symbol table is initialized with the set of **keywords**

```
// global.h
#define NUM 256 // token returned by lexan
#define DIV 257 // division operation token
#define MOD 258 // mod operation token
#define ID 259 // identifier token
```

```
// init.c
insert("div", DIV);
insert("mod", MOD);
```

```
// lexer.c
int lexan()
{
    ...
    tokenval = lookup(lexbuf);
    if (tokenval == 0)
        tokenval = insert(lexbuf, ID);
    return symtable[tokenval];
}
```

Reading Number - Example

factor → (*expr*)
| num { print(num.value) }

```
factor()
{
    if (lookahead == '(')
        { match('('); expr(); match(')');
    }
    else if (lookahead == NUM)
        {printf(" %d ", tokenval); match(NUM);
    }
    else error();
}
```

Reading Identifier - Example

$$\begin{aligned} factor \rightarrow & (\ expr) \\ & | \text{id} \{ \text{print(id.string)} \} \end{aligned}$$

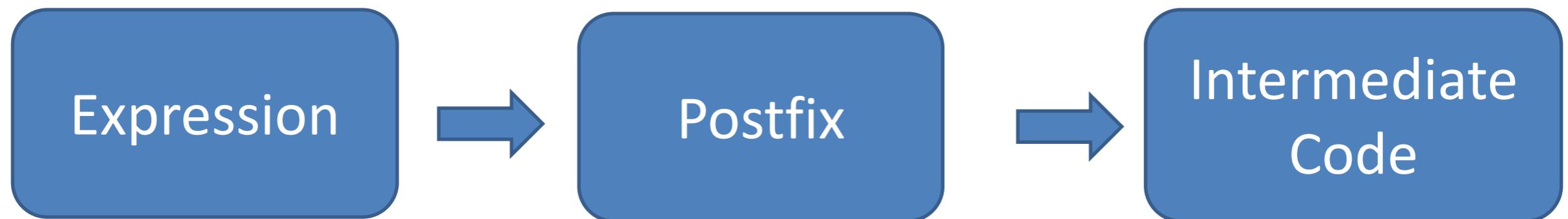
```
factor()
{
    if (lookahead == '(')
    {
        match('('); expr(); match(')');
    }
    else if (lookahead == ID)
    {
        printf(" %s ", symtable[tokenval].lexptr);
        match(ID);
    }
    else error();
}
```

Reading Multi-Number Operations

morefactors → **div** *factor* { print('DIV') } *morefactors*
| **mod** *factor* { print('MOD') } *morefactors*
| ...

```
// parser.c
morefactors()
{   if (lookahead == DIV)
    {   match(DIV) ; factor() ; printf("DIV") ; morefactors() ;
    }
  else if (lookahead == MOD)
  {   match(MOD) ; factor() ; printf("MOD") ; morefactors() ;
  }
  else
    ...
}
```

Intermediate Code for Abstract Stack Machine



Intermediate Code for Abstract Stack Machine

- The front end of the compiler creates an **intermediate representation of the source program** from which the back end generates the target program
- Stack: **last in first out (LIFO)** storage. It uses two operations : **push, pop**
 - **Push** puts an item at the **top of stack**
 - **Pop** retrieves an item **from the top of stack**

Evaluating Expressions Using Stack

- Because a **stack is LIFO**, any operation must **access data item from the top**
- **No addressing** is required because expression is implied in the operators on stack.
- Any **expression** can be transformed into a **postfix order and stack** and evaluated without explicit localizing of any variable.

Generic Instructions for Stack Manipulation

push <i>v</i>	push constant value <i>v</i> onto the stack
rvalue <i>l</i>	push contents of data location <i>l</i>
lvalue <i>l</i>	push address of data location <i>l</i>
pop	discard value on top of the stack
:=	the r-value on top is placed in the l-value below it and both are popped
copy	push a copy of the top value on the stack
+	add value on top with value below it, pop both and push result
-	subtract value on top from value below it, pop both and push result
* , / , ...	same for other arithmetic operations
< , & , ...	same for relational and logical operations

Generic Control Flow Instructions

label *l*

label instruction with *l*

goto *l*

jump to instruction labeled *l*

gofalse *l*

pop the top value, if zero then jump to *l*

gotrue *l*

pop the top value, if nonzero then jump to *l*

halt

stop execution

jsr *l*

jump to subroutine labeled *l*, push returned address

return

pop returned address and return to caller

Evaluating Expressions using Stack - Example

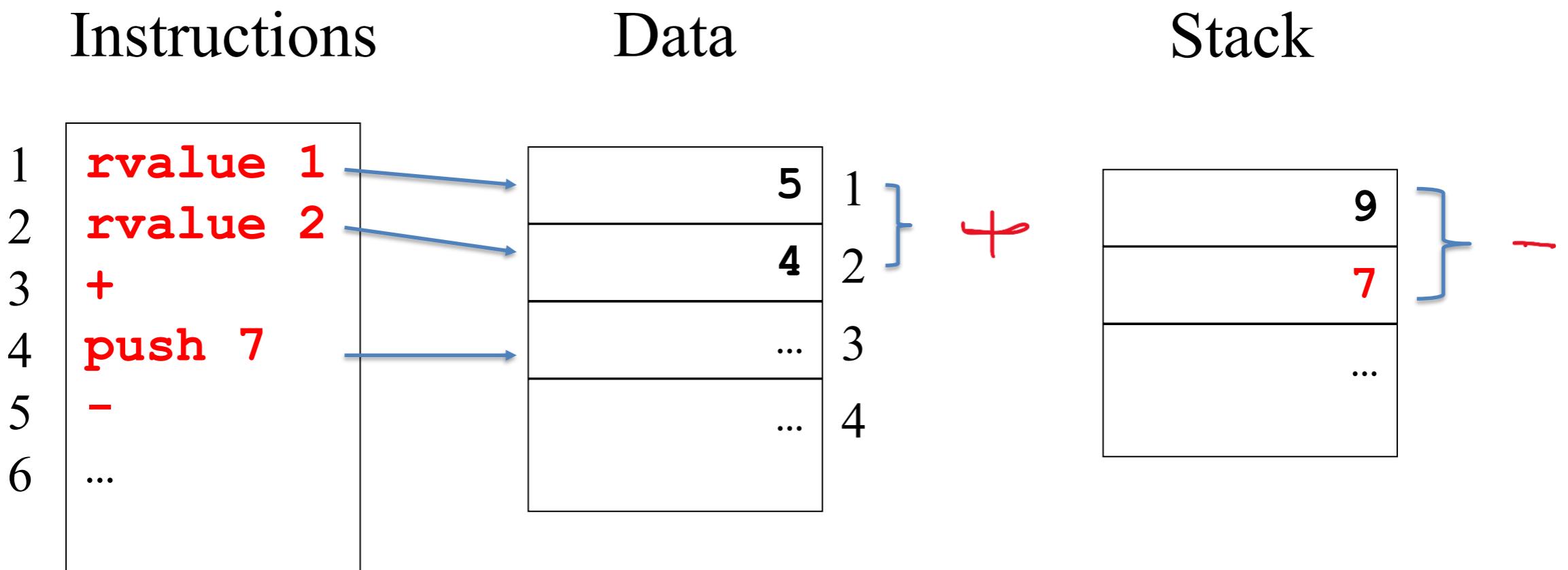
Expression: $B + C - 7$

Postfix: $B\ C\ +\ 7\ -$

Intermediate Code:

```
rvalue B    // push contents of data location B  
rvalue C    // push contents of data location C  
+ // add value on top with value below it,  
   pop both and push result  
push 7    // push constant value 7  
- //subtract value on top from value below it,  
   pop both and push result
```

Abstract Stack Machines - Example



Syntax-Directed Translation of Expressions

$expr \rightarrow term\ rest \{ expr.t := term.t \parallel rest.t \}$

$rest \rightarrow +\ term\ rest_1 \{ rest.t := term.t \parallel '+' \parallel rest_1.t \}$

$rest \rightarrow -\ term\ rest_1 \{ rest.t := term.t \parallel '-' \parallel rest_1.t \}$

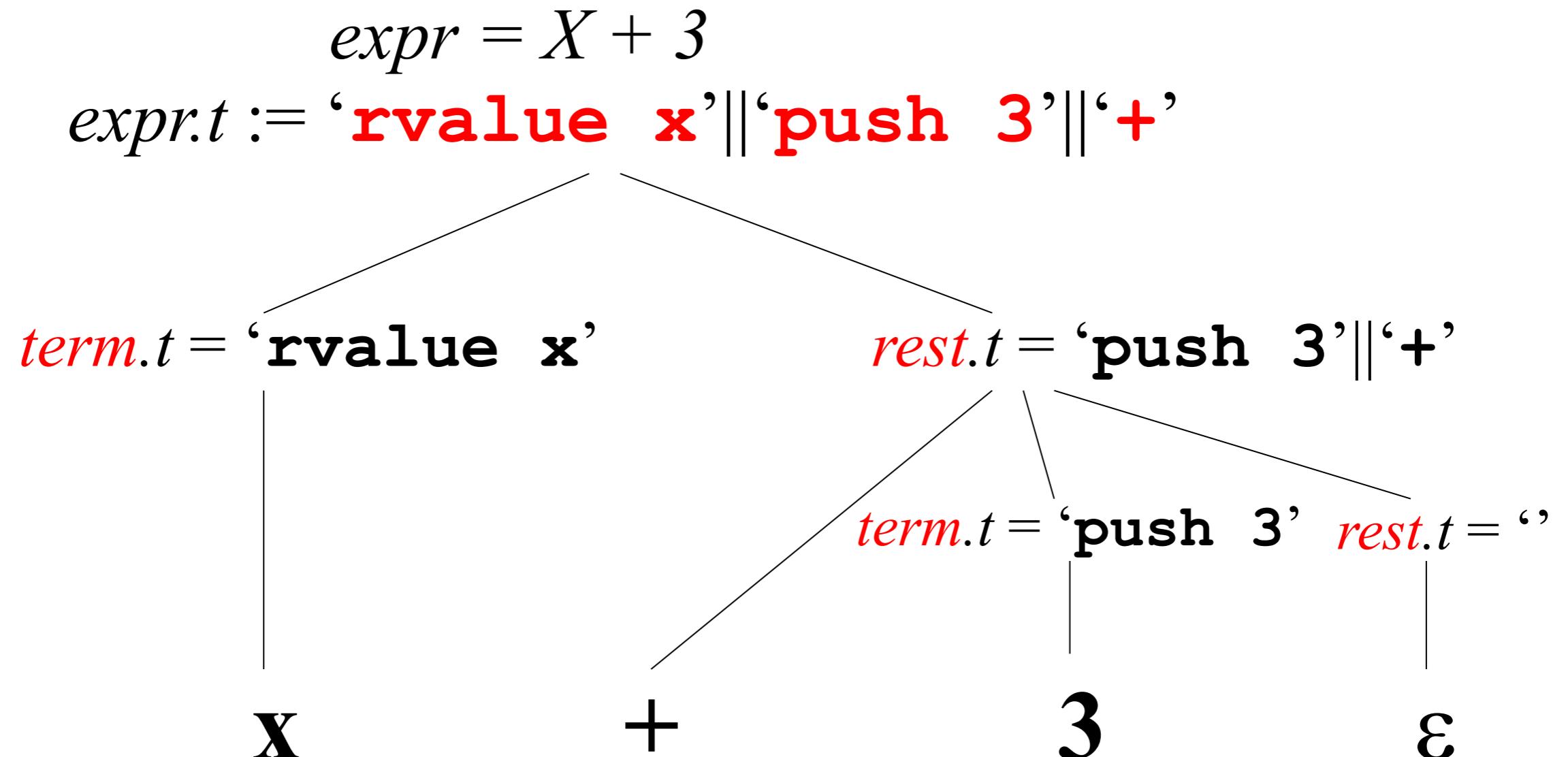
$rest \rightarrow \epsilon \{ rest.t := '' \}$

$term \rightarrow id \{ term.t := 'rvalue' \parallel id.lexeme \}$

$term \rightarrow num \{ term.t := 'push' \parallel num.value \}$

Syntax-Directed Translation of Expressions

Parsing - Example



Translation Scheme to Generate Abstract Machine Code - Example1

expr → *term moreterms*

moreterms → + *term { print(‘+’) } moreterms*

moreterms → - *term { print(‘-’) } moreterms*

moreterms → ε

term → *factor morefactors*

morefactors → * *factor { print(‘*’) } morefactors*

morefactors → div *factor { print(‘DIV’) } morefactors*

morefactors → mod *factor { print(‘MOD’) } morefactors*

morefactors → ε

factor → (*expr*)

factor → num { print(‘push ’ || num.value) }

factor → id { print(‘rvalue ’ || id.lexeme) }

Translation Scheme to Generate Abstract Machine Code - Example2

stmt → **id** = { print('lvalue' || *id.lexeme*) } *expr* { print('=') }

lvalue <i>id.lexeme</i>
code for <i>expr</i>
=

Translation Scheme to Generate Abstract Machine Code - Example3

stmt → if *expr* { *out* = newlabel(); print('gofalse' || *out*) }
then *stmt* { print('label'|| *out*) }

code for <i>expr</i>
gofalse <i>out</i>
code for <i>stmt</i>
label <i>out</i>

Translation Scheme to Generate Abstract Machine Code – Example4

stmt → **while** { *test* = newlabel(); print('label' || *test*) }
expr { *out* = newlabel(); print('gofalse' || *out*) }
do *stmt* { print('goto' || *test* || 'label' || *out*) }

label <i>test</i>
code for <i>expr</i>
gofalse <i>out</i>
code for <i>stmt</i>
goto <i>test</i>
label <i>out</i>