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 Entry

Each entry in the symbol table contains a string and a token value:

```
struct entry
{    char *lexptr; // lexeme (string)
     int tokenvalue;
};
struct entry symtable[];
```

`insert(s,t)`: inserts new entry for string `s` token `t`

`lookup(s)`: returns array index to entry for string `s` or returns 0 if s is not found

The symbol table is initialized with the reserved keywords and their tokens

Possible Symbol table implementations:
- simple C code
- hashtables

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

$$factor \rightarrow \textbf{(}\ expr\ \textbf{)}$$
$$\quad\quad | \ \textbf{num}\ \{\ \text{print}(\textbf{num}.\text{value})\ \}$$

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

Reading Identifier - Example

$$factor \rightarrow \textbf{(}\ expr\ \textbf{)}$$
$$|\ \textbf{id}\ \{\ \text{print}(\textbf{id}.\text{string})\ \}$$

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

Reading Multi-Number Operations

$$
\begin{aligned}
\textit{morefactors} \rightarrow\ & \textbf{div}\ \textit{factor}\ \{\ \text{print('DIV')}\ \}\ \textit{morefactors} \\
|\ & \textbf{mod}\ \textit{factor}\ \{\ \text{print('MOD')}\ \}\ \textit{morefactors} \\
|\ & \ldots
\end{aligned}
$$

```
// 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` *v*	push constant value *v* onto the stack
`rvalue` *l*	push contents of data location *l*
`lvalue` *l*	push address of data location *l*
`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*

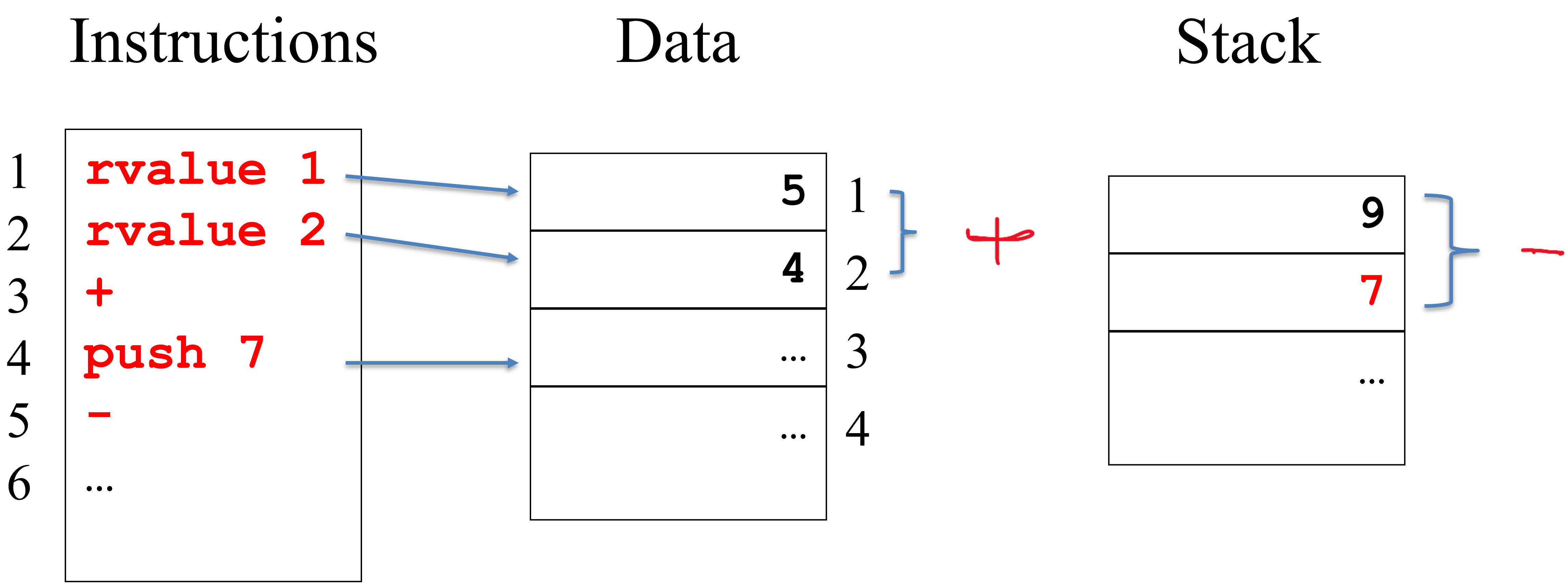
rvalue C // push contents of data location *C*

\+ // 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

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

$$rest \rightarrow +\ term\ rest_1\ \{\ rest.t := term.t \,\|\, \text{‘}\mathbf{+}\text{’} \,\|\, rest_1.t\ \}$$

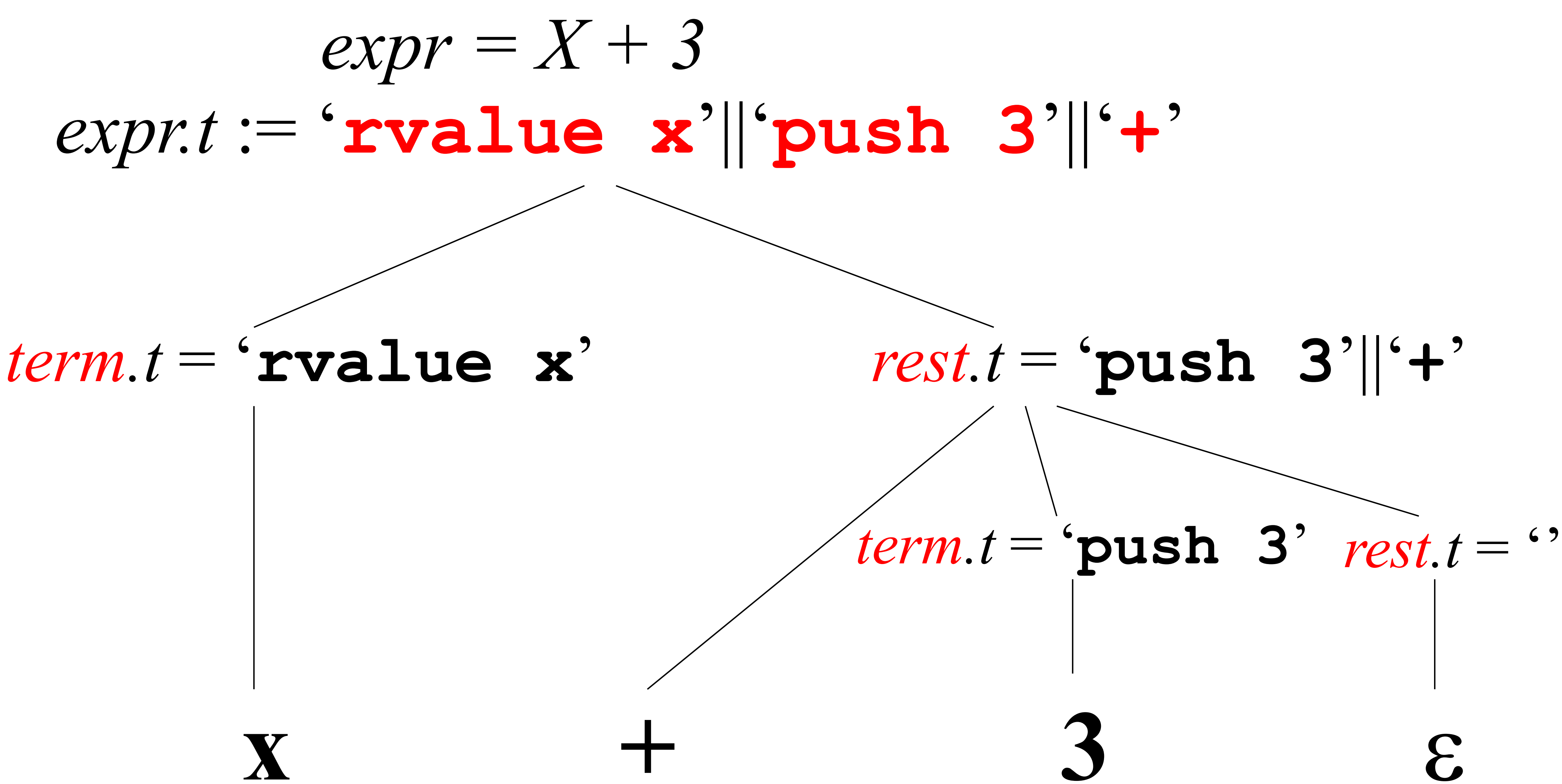
$$rest \rightarrow -\ term\ rest_1\ \{\ rest.t := term.t \,\|\, \text{‘}\mathbf{-}\text{’} \,\|\, rest_1.t\ \}$$

$$rest \rightarrow \varepsilon\ \{\ rest.t := \text{‘’}\ \}$$

$$term \rightarrow \mathbf{id}\ \{\ term.t := \text{‘}\mathtt{rvalue}\text{’} \,\|\, \mathbf{id}.lexeme\ \}$$

$$term \rightarrow \mathbf{num}\ \{\ term.t := \text{‘}\mathtt{push}\text{’} \,\|\, \mathbf{num}.value\ \}$$

Syntax-Directed Translation of Expressions Parsing - Example



Translation Scheme to Generate Abstract Machine Code - Example1

$$\begin{aligned}
\mathit{expr} &\rightarrow \mathit{term}\ \mathit{moreterms} \\
\mathit{moreterms} &\rightarrow +\ \mathit{term}\ \{\ \text{print}(\text{‘}+\text{’})\ \}\ \mathit{moreterms} \\
\mathit{moreterms} &\rightarrow \mathbf{-}\ \mathit{term}\ \{\ \text{print}(\text{‘}\mathbf{-}\text{’})\ \}\ \mathit{moreterms} \\
\mathit{moreterms} &\rightarrow \varepsilon \\
\mathit{term} &\rightarrow \mathit{factor}\ \mathit{morefactors} \\
\mathit{morefactors} &\rightarrow \mathbf{*}\ \mathit{factor}\ \{\ \text{print}(\text{‘}\mathbf{*}\text{’})\ \}\ \mathit{morefactors} \\
\mathit{morefactors} &\rightarrow \mathbf{div}\ \mathit{factor}\ \{\ \text{print}(\text{‘}\mathbf{DIV}\text{’})\ \}\ \mathit{morefactors} \\
\mathit{morefactors} &\rightarrow \mathbf{mod}\ \mathit{factor}\ \{\ \text{print}(\text{‘}\mathbf{MOD}\text{’})\ \}\ \mathit{morefactors} \\
\mathit{morefactors} &\rightarrow \varepsilon \\
\mathit{factor} &\rightarrow \mathbf{(}\mathit{expr}\mathbf{)} \\
\mathit{factor} &\rightarrow \mathbf{num}\ \{\ \text{print}(\text{‘}\mathbf{push}\ \text{’} \parallel \mathbf{num}.\mathit{value})\ \} \\
\mathit{factor} &\rightarrow \mathbf{id}\ \{\ \text{print}(\text{‘}\mathbf{rvalue}\ \text{’} \parallel \mathbf{id}.\mathit{lexeme})\ \}
\end{aligned}$$

Translation Scheme to Generate Abstract Machine Code - Example2

stmt $\rightarrow$ **id** = { print('**`lvalue`**' || **id**.*lexeme*) } *expr* { print('**`=`**') }

`lvalue` **id**.*lexeme*
code for *expr*
`=`

Translation Scheme to Generate Abstract Machine Code - Example3

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

code for *expr*
`gofalse` *out*
code for *stmt*
`label` *out*

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` *test*
code for *expr*
`gofalse` *out*
code for *stmt*
`goto` *test*
`label` *out*