Parsing Simple Programs

Prof. Naga Kandasamy
ECE Department, Drexel University

We discuss how to parse simple programs comprising of assignment statements, as well as selection and iterative statements. We will also discuss how to generate code for the stack-based virtual machine.

Assignment Statements

Consider the following code snippet.

```
a = 3;
b = 10;
c = a + (b - 4)/2;
```

The following grammar will correctly parse a program comprising entirely of assignment statements, of the form shown above. We add a production rule involving `identifier` to the list of rules under *Factor* to accommodate symbols occurring within expressions on the right-hand side of the assignment statement.

Program	$\rightarrow$	*StatementList* EOF
statementList	$\rightarrow$	*Statement*
	$\mid$	*Statement StatementList*
Statement	$\rightarrow$	*AssignmentStmt*
AssignmentStmt	$\rightarrow$	`identifier` ASSIGN *Expr* SEMI
Expr	$\rightarrow$	*Term* PLUS *Expr*
	$\mid$	*Term* MINUS *Expr*
	$\mid$	*Term*
Term	$\rightarrow$	*Factor* MULTIPLY *Term*
	$\mid$	*Factor* DIVIDE *Term*
	$\mid$	*Factor*
Factor	$\rightarrow$	PLUS *Factor*
	$\mid$	MINUS *Factor*
	$\mid$	LPAREN *Expr* RPAREN
	$\mid$	`number`
	$\mid$	`identifier`

Tokens appearing in the foregoing grammar are:

```
(PLUS, `+'}
(MINUS, `-')
(MUL, `*'}
(DIV, `/')
(LPAREN, `(')
(RPAREN, `)')
(SEMI, `;')
(ASSIGN, `=')
```

The `EOF` symbol denotes end of file.

The abstract syntax tree (AST) generated by the parser for our code snippet is shown below.

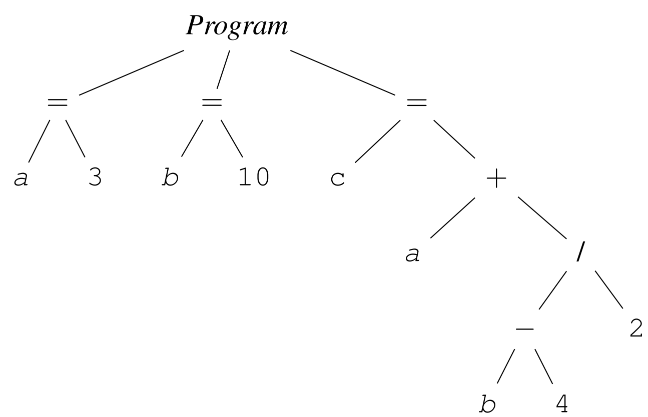


Fig. 1: The abstract syntax tree corresponding to the code snippet shown earlier.

In addition to the binary and unary operator nodes introduced in the earlier lecture, we define new nodes within the AST to represent the program, assignment statement, and the identifier.

```
class Program(AST):
"""Node represents a program. It contains a list of assignment-statement
nodes as its children.
"""
def __init__(self):
    self.children = []      # List of statements

class Assign(AST):
"""Node represents an assignment statement. The left edge stores a
variable node and right edge stores a node returned by the expr() method.
"""
def __init__(self, left, op, right):
    self.left = left
    self.token = self.op = op
    self.right = right

class Identifier(AST):
"""Node represents an identifier."""
def __init__(self, token):
    self.token = token
    self.value = token.value
```

Code Generation: Assume our example code is stored in a text file called *foo.txt*.

```
// foo.txt
a = 3;
b = 10;
c = a + (b - 4)/2;
```

Generating code for our program from the AST is straightforward. We use a *symbol table* to track various variables within the source code. Visiting the children of the *Program* node from left to right, upon arriving at an assignment node, the variable occurring on the left-hand side of the assignment operator is added to the symbol table, if it has not previously, along with the corresponding RAM address location. Variables are stored in the local segment. Contents of the symbol table after the three nodes have been visited are as follows.

Variable name	Base	Offset
a	LCL	0
b	LCL	1
c	LCL	2

When a variable is encountered on the right-hand side of the assignment-operator node, we perform a lookup into the symbol table to locate the corresponding address. If the variable does not occur in the symbol table, it is undefined, leading to an error.

VM commands for the program can be generated by vising children of the *Program* node from left to right. When visiting the assignment node, code for the right-hand side (the expression) is generated first, followed by assignment to the identifier.

```
// foo.vm

// Initialize SP and LCL
set SP 256
set LCL 16

// a = 3
push constant 3
pop local 0

// b = 10
push constant 10
pop local 1

// c = a b 4 - 2 / +
push local 0
push local 1
push constant 4
sub
push constant 2
call div 2
add
pop local 2
```

In the above, we assume the availability of a function called *div* which performs integer division on two operands.

Boolean Expressions

Results of Boolean and relational operators produce Boolean values — that is, `True` or `False`. A common use for Boolean and relational expressions is to alter the program's control flow. The standard expression grammar augmented with Boolean and relational operators is as follows:

Expr	→	*Term* PLUS *Expr*
	|	*Term* MINUS *Expr*
	|	*Term*
Term	→	*Factor* MULTIPLY *Term*
	|	*Factor* DIVIDE *Term*
	|	*Factor*
Factor	→	PLUS *Factor*
	|	MINUS *Factor*
	|	LPAREN *Expr* RPAREN
	|	`number`
	|	`identifier`
BooleanExpr	→	*AndTerm* OR *BooleanExpr*
	|	*AndTerm*
AndTerm	→	*RelationalExpr* AND *AndTerm*
	|	*RelationalExpr*
RelationalExpr	→	*Expr* LT *RelationalExpr*
	|	*Expr* LE *RelationalExpr*
	|	*Expr* EQ *RelationalExpr*
	|	*Expr* NE *RelationalExpr*
	|	*Expr* GE *RelationalExpr*
	|	*Expr* GT *RelationalExpr*
	|	NOT *RelationalExpr*
	|	*Expr*

New tokens appearing in the grammar are:

```
(NOT, `!'), (OR, `||'), (AND, `&&'), (LT, `<'), (LE, `<='), (EQ, `=='),
(NE, `!='), (GE, `>='), (GT, `>')
```

As is customary, we assume that OR and AND operators are left-associative, and that OR has lowest precedence, then AND, then NOT.

Selection Statements

Consider the following straightforward grammar for the `if-else` statement.

statementList	$\rightarrow$	*Statement*
	$\mid$	*Statement StatementList*
Statement	$\rightarrow$	*CompoundStmt*
	$\mid$	*AssignmentStmt*
	$\mid$	*IfStmt*
	$\mid$	*...other statements...*
CompoundStmt	$\rightarrow$	LCURLY *statementList* RCURLY
IfStmt	$\rightarrow$	`if` LPAREN *BooleanExpr* RPAREN *Statement* `else` *Statement*
	$\mid$	`if` LPAREN *BooleanExpr* RPAREN *Statement*

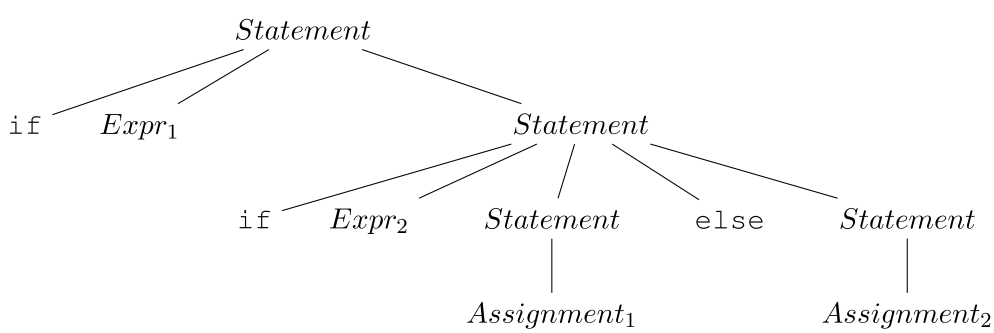
The identifiers, `if`, `else`, and `while` are now keywords in our programming language. New tokens appearing in the grammar are:

`(LCURLY, `{'), (RCURLY, `}')`

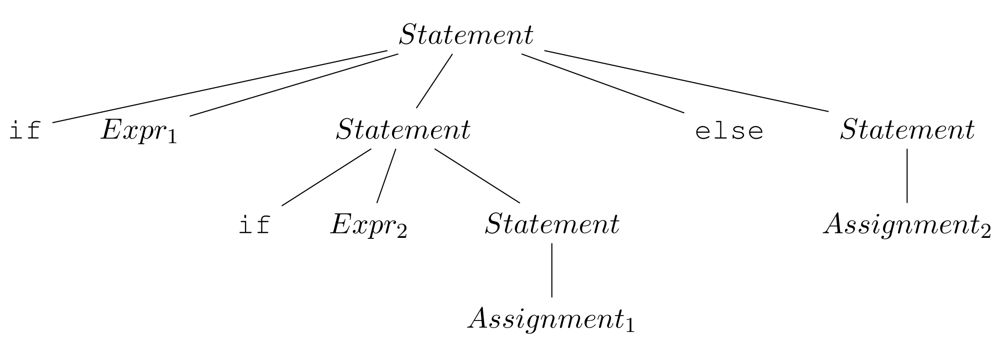
The grammar shows that the `else` clause is optional. Unfortunately, this grammar is *ambiguous.* For example, the code fragment

`if` ($Expr_1$) `if` ($Expr_2$) $Assignment_1$ `else` $Assignment_2$

has two parse trees. The first derivation has $Assignment_2$ controlled by the inner `if` as shown by the following parse tree, such that $Assignment_2$ executes if $Expr_1$ is true and $Expr_2$ is false.



The second derivation associates the `else` clause with the outer `if` so that $Assignment_2$ executes when $Expr_1$ is false, independent of the value of $Expr_2$.



Clearly, these two derivations produce different behaviors in the compiled code. In all programming languages with conditional statements of this form, the first parse tree is preferred. The general rule is, to match each `else` with the closest unmatched `if`. This disambiguating rule can be incorporated directly into the following modified grammar:

Statement	$\rightarrow$	*CompoundStmt*
	$\mid$	*AssignmentStmt*
	$\mid$	*IfStmt*
	$\mid$	*...other statements...*
IfStmt	$\rightarrow$	`if` LPAREN *BooleanExpr* RPAREN *WithElse* `else` *Statement*
	$\mid$	`if` LPAREN *BooleanExpr* RPAREN *Statement*
WithElse	$\rightarrow$	`if` LPAREN *BooleanExpr* RPAREN *WithElse* `else` *WithElse*
	$\mid$	*AssignmentStmt*

The grammar now includes a rule which determines which `if` controls an `else`, ensuring that each `if` has an unambiguous match to a specific `else` — bind each `else` to the innermost unclosed `if`. The above grammar has only one derivation. For example,

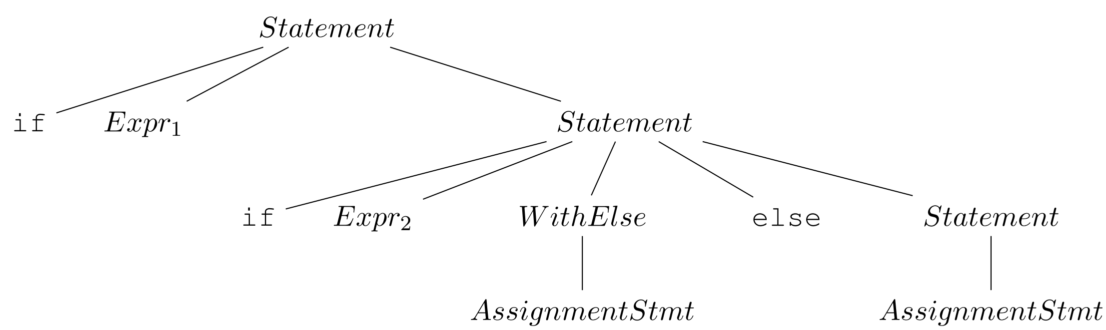
`if` ($Expr_1$) *Statement*

`if` ($Expr_1$) `if` ($Expr_2$) *WithElse* `else` *Statement*

`if` ($Expr_1$) `if` ($Expr_2$) *WithElse* `else` *AssgnmentStmt*

`if` ($Expr_1$) `if` ($Expr_2$) *AssignmentStmt* `else` *AssgnmentStmt*

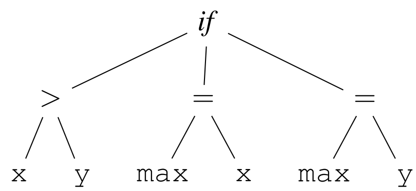
The corresponding parse tree is as follows.



Consider the following snippet of code involving an *if-else* statement.

```
if (x > y)
    max = x;        // Statement 1
else
    max = y;        // Statement 2
```

The node for the *if-else* construction within the AST can be constructed as follows.



Iterative Statements

It is straightforward to add support in our grammar for loops such as the following:

```
for (int i = 0; i < 100; i++) {
    // Code block
}

while (i < n) {
    // Code block
}
```

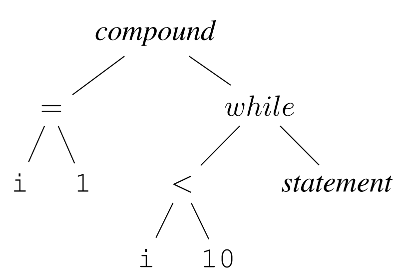
The following rules are added to our grammar to support `for` and `while` statements.

Statement	→	*CompoundStmt*
	|	*AssignmentStmt*
	|	*ForStmt*
	|	*WhileStmt*
ForStmt	→	`for` LPAREN *AssignmentStmt* SEMI *RelationalExpr* SEMI *AssignmentStmt* RPAREN *Statement*
WhileStmt	→	`while` LPAREN *BooleanExpr* RPAREN *Statement*

Finally, consider the following snippet of code involving a *while* loop.

```
{
    i = 1;
    while (i < 10)  {
        // Code block
        i++;
    }
}
```

The AST generated for the code by the recursive descent parser is as follows.



To generate VM commands from the AST, we visit each child under the compound-statement parent node from left to right. Commands for the assignment-operator node are generated as follows, assuming variables reside in the local segment.

```
// VM commands to evaluate expression on RHS, in postfix form, and place
// result on working stack
pop local 0
```

VM commands for the while-statement node are generated along the following lines.

```
label L1
// VM commands to evaluate expression and place result on stack
not             // Obtain NOT(expression)
if-goto L2
// VM comamnds for statement
goto L1

label L2
// Rest of code
```

Recall that the *if-goto* command pops the top element from the stack, and if the element is non-zero, it transfers control to the labeled statement.

Subroutines

Procedures, also called subroutines or functions, are the basic unit of work for a compiler. A typical compiler processes a collection of procedures and produces code for them that will link and execute correctly with other collections of compiled procedures. A fully-implemented parser also operates at the granularity of procedures, parsing each procedure in the program one at a time as shown by the top-level parse tree below. Each node contains within it the following information: return type, procedure name, arguments passed to the procedure, ASTs for the statements contained within the procedure's body, and a symbol table that stores the name and type of local variables declared within the procedure's scope. *Note that main() must be one of the functions, since it is the entry point into the program.*

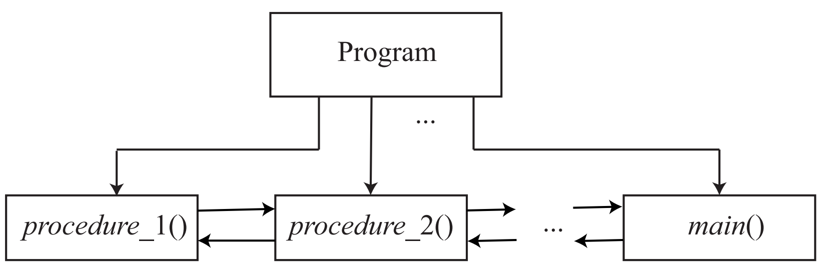


Fig. 2: Program as a collection of procedures.

The grammar for the parser is listed below where ϵ and `EOF` denote an empty string and the end of file, respectively.

Program	$\rightarrow$	*ProcedureList* EOF
ProcedureList	$\rightarrow$	*Procedure ProcedureList*
	$\mid$	*Procedure*
Procedure	$\rightarrow$	*ReturnValue* `identifier` LPAREN *ParameterList* RPAREN *CompoundStmt*
	$\mid$	*ReturnValue* `identifier` LPAREN ϵ RPAREN *CompoundStmt*
ParameterList	$\rightarrow$	*DataType* `identifier` *MoreParams*
MoreParams	$\rightarrow$	COMMA *DataType* `identifier` *MoreParams*
	$\mid$	ϵ
CompoundStmt	$\rightarrow$	LCURLY *statementList* RCURLY
statementList	$\rightarrow$	*Statement*
	$\mid$	*StatementList Statement*
Statement	$\rightarrow$	*CompoundStmt*
	$\mid$	*AssignmentStmt*
	$\mid$	*FunctionCall*
	$\mid$	*ReturnStmt*
	$\mid$	ϵ
AssignmentStmt	$\rightarrow$	*DataType* `identifier` ASSIGN *Expr* SEMI
	$\mid$	`identifier` ASSIGN *Expr* SEMI
FunctionCall	$\rightarrow$	`identifer` LPAREN *ArgList* RPAREN SEMI
	$\mid$	`identifier` LPAREN ϵ RPAREN SEMI
ArgList	$\rightarrow$	*Expr MoreArgs*
MoreArgs	$\rightarrow$	COMMA *Expr MoreArgs*
	$\mid$	ϵ
ReturnStmt	$\rightarrow$	`return` *Expr* SEMI
	$\mid$	`return` SEMI
ReturnValue	$\rightarrow$	`void`
	$\mid$	`int`
	$\mid$	`float`
DataType	$\rightarrow$	`int`
	$\mid$	`float`