The Structure of a Compiler

A ***compiler*** comprises three main modules:

*A **lexical analyser*** or ***scanner***: This groups characters into words (operators, variable names, constants, etc.), called ***tokens*** or ***lexemes***. (A lexical analyser can be generated using *Lex*.)

*A **parser***: This gathers the words into phrases, such as expressions, statements, etc. Its output is frequently expressed as a ***syntax tree***. A syntax tree shows the structure of the source program, e.g., its division into procedures, statements, statements within statements, expressions, and so on.

*A **code generator***: This takes the syntax tree and generates ***machine code***.

Scope of the Exercise

To keep the example short, we shall only consider the evaluation of arithmetic expressions. We allow for the usual *operators*: +, -, *, /, **, (, and). (They operate on *operands*.)

An example input is,

`1*2+3/4-5/6**-(7-8)`

Applying the *BEDMAS* rules, the expression is equivalent to the fully bracketed form,

((1 x 2) + (3 ÷ 4)) – (5 ÷ (6 (- (7 – 8))

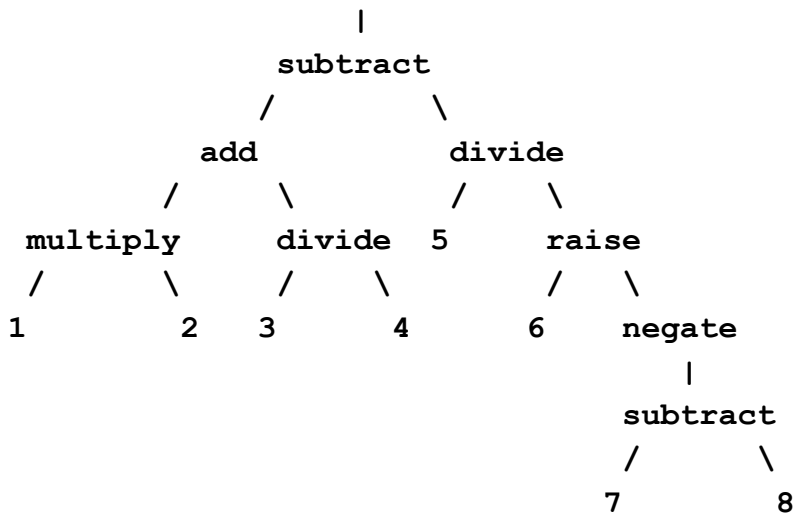
But as far as the computer is concerned, this is just a sequence of ASCII codes:

`49,42,50,43,51,47,52,45,53,47,54,42,42,45,40,55,45,56,41`

The *scanner* groups the codes into *tokens*, classifies them, and adds an **end** marker:

```
integer(1), times, integer(2), plus, integer(3), over,
integer(4), minus, integer(5), over, integer(6), power, minus,
open, integer(7), minus, integer(8), close, end
```

The *parser* then structures the tokens into levels, according to the *BEDMAS* rules:



where the top level operator is *subtract*, containing *add* and *divide* at the next level, and so on. This diagram is the *syntax tree* of the expression. (The parser distinguishes a ***unary*** *minus* (*negate*) from an ***infix*** *minus* (*subtract*).)

Before we can apply an *operator*, we must evaluate its *operands*: The first step is to *multiply* 1 by 2. The next is to *divide* 3 by 4. Only then can we *add* 2 and 0.75. The order of evaluation is therefore bottom to top, called *reverse Polish notation*:

1, 2, *multiply*, 3, 4, *divide*, *add*, 5, 6, 7, 8, *subtract*, *negate*, *raise*, *divide*, *subtract*.

Evaluation is most easily done using a last-in, first-out store, called a ***stack***. Starting with an empty stack, **[ ]**, we ***push*** the first operand onto the stack, giving **[1].** The next step is to *push* the second operand to the stack, giving **[2,1]**. Elements are added onto the ***top*** of the stack (which we show on the *left*). We then ***pop*** the top two elements from the stack,

multiply them, and *push* the result back onto the stack, giving **[2]**. We then *push* 3 (giving **[3,2]**) and 4 onto the stack, giving **[4,3,2]**.

To apply the *divide* operation, we *pop* the top two operands, divide, and *push* the result back onto the stack, giving **[0.75,2]**. Notice two things: First, although the stack contained **[4,3,2]**, we had to be careful to evaluate 3÷4, not 4÷3. Second, the rest of the stack (**2**) remained undisturbed.

You are invited to check that the remaining operations yield **[2.75]**, **[5,2.75]**, **[6,5,2.75]**, **[7,6,5,2.75]**, **[8,7,6,5,2.75]**, **[-1,6,5,2.75]**, **[1,6,5,2.75]**, **[6.0,5,2.75]**, **[0.8333333333333337, 2.75]** and **[1.9166666666666665]** successively.

From this, we see that the evaluation strategy is:

If it is an *operand*, *push* it onto the stack.

If it is an *operator*, *pop* its operands, execute the operation, and *push* the result on the stack.

We can either do these operations immediately, making a desk calculator, or generate *machine code* that will do them later.

Code Generation

The code we need to generate depends on the architecture of the target machine. Some computers have a stack-based architecture with machine codes that push operands, and operators that operate on the top two stack locations. Others have several registers which can be used as a stack. Here we assume the least convenient set up: a computer with a single *accumulator* register. The stack is implemented using temporary variables in RAM: `STACK1, STACK2, … STACK`*n*, where *n* varies with the number of items currently in the stack. `STACK1` is always the *bottom* of the stack. The top of the stack is always in the *accumulator*, and the next-to-top element is in the highest numbered `STACK`*n*.

```
OPCODE   OPERAND  COMMENTS
LDAI     7        Load the accumulator with the value 7.
SUAI     8        - : Subtract 8 from the accumulator.
MUAI     -1       unary - : Negate the accumulator.
STA      STACK1   ** : Push the 2nd operand onto the stack. Evaluate 1st operand ...
LDAI     6        Load the accumulator with the value 6.
CALL     LN       Find the log of the 1st operand.
MUA      STACK1   Multiply the log by the 2nd operand and pop the stack.
CALL     EXP      Find the anti-log of the product.
STA      STACK1   / : Push the 2nd operand onto the stack. Evaluate 1st operand ...
LDAI     5        Load the accumulator with the value 5.
DVA      STACK1   Divide the 1st operand by the second operand and pop the stack.
STA      STACK1   - : Push the 2nd operand onto the stack. Evaluate 1st operand ...
LDAI     1        Load the accumulator with the value 1.
MUAI     2        * : Multiply the accumulator by 2.
STA      STACK2   + : Push the 1st operand onto the stack. Evaluate 2nd operand ...
LDAI     3        Load the accumulator with the value 3.
DVAI     4        / : Divide the accumulator by 4.
ADA      STACK2   + (contd): Add the 2nd operand to the 1st and pop the stack.
SUA      STACK1   Subtract the 2nd operand from the 1st operand.
CALL     PRINT    Display the value of the accumulator.
```

Note that `MUAI` multiplies the accumulator by the *value* of its operand, whereas `MUA` multiplies it by the *contents* of its operand. (Similarly for `SUAI`, `SUA`, etc.)

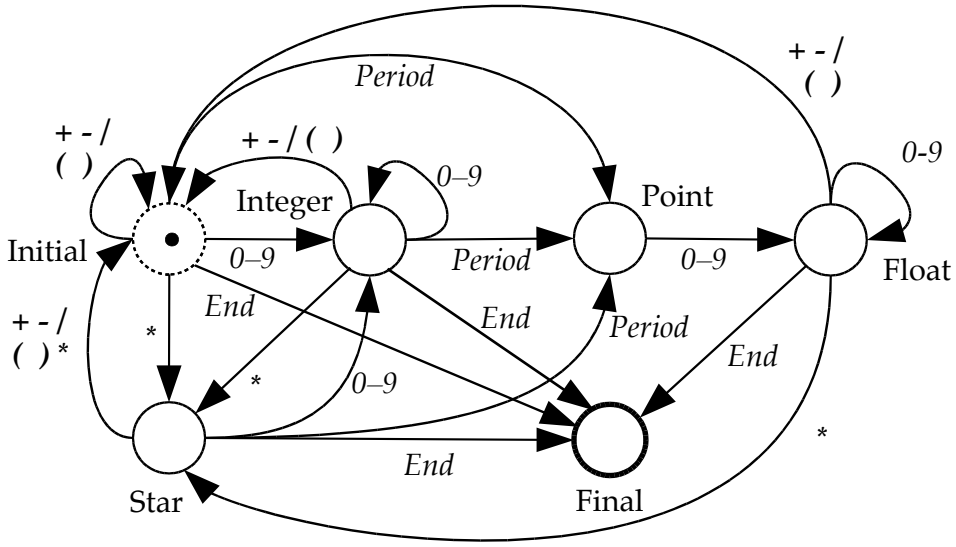
Notice also how the generated code for an operator can be split by the code needed to evaluate its operands. The outermost *subtract* spans seven instructions (by pushing down its second operand from the accumulator to `STACK1`) before the `SUA` is executed at the last instruction but one! This sort of thing makes machine code hard to understand and harder still to write. (In practice, reading machine code is worse than this, because compilers generate binary code, not assembler, and they don't write helpful comments.)

How Does the Scanner Work?

A scanner is typically built around a ***finite-state automaton*** (**FSA**). The ***state*** of an FSA is usually represented by the value of a variable (typically called *State*).

We begin by defining a pattern for each kind of token we want to recognise. We can do this using ***regular expressions*** (as used by *grep*). For example, we can define a *float* by the expression [0-9]*.[0–9]+, meaning zero or more digits, followed by a period, followed by one or more digits. The FSA for this token needs four states: *initial, integer, point* and *float*. The FSA starts in its *initial* state. If a digit is read from the input, the FSA moves to the *integer* state but a period (.) moves the FSA to the *point* state. In the *integer* state, any additional digits cause the FSA to remain in the *integer* state, but a decimal point moves it to the *point* state. A digit moves the *point* state to the *float* state; anything else would be an error. In the *float* state, any non-digit character marks the end of the *float*. These rules allow *floats* such as '1.5' and '.5', but treat '1.' as an error.

For the purposes of this example, only two more states are needed: *star*, which means that one '*' has been read, but we are not yet certain if it is the first of a pair, '**', and *final*, which occurs at the end of the input.



The above ***state transition diagram*** and the following ***state transition matrix*** show the new states that follow each ***transition*** the FSA can make. (Those marked with a question mark should never happen, and the *transitions* are attempts at error recovery.)

Input**State**	***initial***	***integer***	***point***	***float***	***star***
digit (0–9)	*integer*	*integer*	*float*	*float*	*integer*
period	*point*	*point*	*point?*	*initial?*	*point*
+, -, /, (, or)	*initial*	*initial*	*initial?*	*initial*	*initial*
*	*star*	*star*	*star?*	*star*	*initial*
end of input	*final*	*final*	*final?*	*final*	*final*
anything else	*initial?*	*initial?*	*initial?*	*initial?*	*initial?*

(An *FSA* can be derived automatically from the regular expressions for each kind of token, as indeed it is by *grep* or *lex*. Explaining *how* would be a talk in itself.)

For example, the sequence '1.2*34' would yield the following sequence of states:

initial, integer, point, float, star, integer, integer, final.

To do anything useful, we must associate a sequence of ***actions*** with each transition. For example, when a digit is read, as in the transition from *integer* to *integer*, we add a digit character to the growing token. When we reach the end of an integer, as in the last transition, from *integer* to *final*, we convert the token's character string to a binary integer, and add it to the scanner's output stream.

(In the program I have written, these two particular transitions are defined by,

```
integer >== digit/[append] ==> integer.
integer >== terminator/[integer, end] ==> final.
```

where the *append* action appends the digit to the growing token, *integer* converts the token to binary form, and *end* creates an end marker.)

Finite-state automata are powerful, but can't deal with the nested structures that can arise in most programming languages, such as the nesting of sub-expressions within brackets.

How Does the Parser Work?

There are several kinds of parser. We shall look at a ***predictive parser***.

A parser is controlled by a set of ***productions***, called a ***grammar***. Each production specifies one of the ways a ***phrase*** can be expanded. The whole input is called a ***sentence*** of the grammar. Here are the productions for our expression ***language***. It uses three kinds of terms: Items in *italics* are the names of *phrases*, such as *expression*. Items in **bold** are the names of *tokens* identified by the scanner, such as **plus**. Items in parentheses, such as '(add)', are *actions*, which we shall ignore for the moment.

sentence ==> [*expression*, **end**]
expression ==> [*term*, *more_terms*]
term ==> [*factor*, *more_factors*]
factor ==> [*atom*, *more_atoms*]
atom ==> [**plus**, *primitive*]
atom ==> [**minus**, *primitive*, (negate)]
atom ==> [*primitive*]
primitive ==> [**integer**(X), (push(X))]
primitive ==> [**float**(X), (push(X))]
primitive ==> [**open**, *expression*, **close**]
more_terms ==> [**plus**, *term*, (add), *more_terms*]
more_terms ==> [**minus**, *term*, (subtract), *more_terms*]
more_terms ==> []
more_factors ==> [**times**, *factor*, (multiply), *more_factors*]
more_factors ==> [**over**, *factor*, (divide), *more_factors*]
more_factors ==> []
more_atoms ==> [**power**, *atom*, *more_atoms*, (raise)]
more_atoms ==> []

From the list of productions, we see that a *sentence* consists of an *expression* followed by an **end** marker. An *expression* consists of a *term*, followed by *more_terms*. *Terms* are sub-expressions linked by **plus** or **minus** operators, as defined by the three productions for *more_terms*. (The third says that *more_terms* can be *empty*; there might be no further *terms*.) In a similar way, *terms* consist of *factors* linked by **times** and **over** operators. *Factors* consist of *atoms* (for lack of a better name) linked by **power** operators, and finally *atoms* consist of a *primitive* (again, for lack of a better name) optionally preceded by a unary **plus** or **minus** operator. A *primitive* is an **integer**, **float** or a parenthesised subexpression.

If we now consider the *actions*, such as push(X), add, etc., we can see that they describe the same set of stack operations we needed earlier to evaluate the expression — or to generate

a parse tree.

Consider the expression "1+2". The scanner renders this as [**integer**(1), **plus**, **integer**(2), **end**]. The token currently being examined by the parser is called its ***look-ahead***, because although it is currently visible to the parser, it hasn't been used yet.

A *predictive parser* initially expands *sentence* as [*expression*, **end**]. It then expands *expression* as [*term*, *more_terms*], giving [*term*, *more_terms*, **end**]. The next few steps are:

[*factor, more_factors*, *more_terms*, **end**]

[*atom*, *more_atoms, more_factors*, *more_terms*, **end**]

[*primitive*, *more_atoms, more_factors*, *more_terms*, **end**]

Since there are three expansions for *primitive*, the parser must choose the right one. Seeing **integer**(1) as the *look-ahead*, it chooses the third production:

[**integer**(1), (push(1)), *more_atoms, more_factors*, *more_terms*, **end**]

At this point the parser consumes **integer**(1) and *pushes* 1 onto the evaluation stack and a stack that builds the tree, leaving **plus** as the look-ahead, and the *prediction* becomes,

[*more_atoms, more_factors*, *more_terms*, **end**]

The look-ahead being **plus**, the next two moves have to be,

[*more_factors*, *more_terms*, **end**]

[*more_terms*, **end**]

The parser then chooses the first production for *more_terms*, giving the prediction,

[**plus**, *term*, (add), *more_terms*, **end**]

It then consumes **plus**, leaving **integer**(2) as the look-ahead. The next few moves are,

[*term*, (add), *more_terms*, **end**]

[*factor, more_factors*, (add), *more_terms*, **end**]

[*atom*, *more_atoms, more_factors*, (add), *more_terms*, **end**]

[*primitive, more_atoms, more_factors*, (add), *more_terms*, **end**]

The look-ahead being **integer**(2), the parser chooses the third production for *primitive*,

[**integer**(2), (push(2)), *more_atoms, more_factors*, (add), *more_terms*, **end**]

The parser consumes **integer**(2) by *pushing* 2 onto the two stacks, leaving **end** as the look-ahead, and the *prediction*,

[*more_atoms, more_factors*, (add), *more_terms*, **end**]

The next few moves are,

[*more_factors*, (add), *more_terms*, **end**],
[(add), *more_terms*, **end**],
[*more_terms*, **end],**
[**end**]

which — the parser having finally added the operands and built an *add*(1,2) parse tree — neatly matches the **end** look-ahead.

The basic action of a *predictive* parser is therefore:

1. If it is a *phrase*, expand it in the light of the look-ahead!
2. If it is an (action), do it!
3. If it is a **token** that matches the look-ahead, consume it!
4. If the token doesn't match, flag an error and attempt error-recovery.

The Code Generator

The code generator could create code to mimic the stack operations exactly as we described earlier, but the example program actually produces better code. We noted that we have to be careful when we subtract, divide or exponentiate to take the operands in the right order. This means that it is sometimes leads to shorter code if we evaluate their

second operands first. Also in the case of multiply and add, their operands ***commute*** (1+2=2+1), so if either operand is just a number, it pays to take it last. This is why it is a good idea to build the parse tree. It allows us to look at the bigger picture. You may wish to compare the example in this text with the result from the example program.

The Unix Tools

Unix provides several tools for compiler construction: *Lex* generates a scanner given a set of regular expressions for the various tokens used by the language. *Yacc* generates a parser given a set of productions, which allow actions to be embedded.

It is also possible to interface a parse tree with the code generation stage of the *C* compiler. However, this seems to be complicated, so it can be easier to generate *C* code and let *C* compile code suited to each particular machine.

Some Tricky Stuff

The *productions* conceal some subtle points:

How do we know that **times** and **over** have greater priority than **plus** and **minus**? Because *expressions* consist of *terms* that consist of *factors* that consist of *atoms*, etc. Had *factors* consisted of *terms*, **plus** and **minus** would have had priority over **times** and **over**.

How do we know that 6-2-3 means (6-2)-3=1 (*left-associative*) rather than 6-(2-3) (*right-associative*), but that 2**3**4 means 2**(3**4)=2^{81}, not (2**3)**4=8^4? Because the grammar places 'add' before *more_terms*, but it places 'raise' after *more_atoms*.

Notice that the grammar is ***recursive***: *primitive* is defined in terms of *expression*, and *expression* is (ultimately) defined in terms of *primitive*.

How does the parser know which production to choose? Because a *parser generator* (me in this case) has pre-computed the possible look-aheads (called ***director sets***). (Again, explaining *how* would fill another talk.) In the example program, the director sets are shown to the right of each production as follows,

```
atom  ==> [plus,primitive]            /[plus].
atom  ==> [minus,primitive,(negate)]  /[minus].
atom  ==> [primitive]                 /[open,float(_),integer(_)].
```

Not all parsers are *predictive:*

Prolog (with which I wrote the example program) uses an ***operator grammar*** that associates each operator with a ***priority*** so it knows in which order to apply them. Such grammars have their limitations: for example, they can't parse **if**…**else**… statements. On the positive side, *Prolog* lets a programmer add new operators to the language. (I added '**>==**' and '**==>**' to my program, to make the transitions and productions more readable.)

Yacc is a ***bottom up*** (*LR*) parser generator. Instead of *expanding* the left sides of productions, it waits until it has matched a right-hand side (using an *FSA*), then *condenses* it to a *phrase*. (It still needs a *look-ahead*, to know when a right-hand side is complete.) *Yacc* also uses operator priorities, to make the grammar more concise and easier to write.

By and large, a compiler for a complete programming language is just more of the same. There will be coding strategies for structures such as **if**…**else**, **while**…**do**, procedure call, and so on. However, there is an important omission from this example: the ***symbol table***. A compiler must keep track of the names (and usually types) given to variables, functions, procedures, etc. These will typically become associated with storage addresses, which are *resolved* when reference is made to them. By looking at the symbol table, the compiler can convert a name into a storage address. Most languages allow names to be *overloaded*, for example, a function can define a local integer *X*, but *X* can also be a global string outside the function. A reference to *X* would normally concern the most local definition: the one with the narrowest *scope*. Compilers typically store a stack of symbol tables, one for each procedure or function, with the most local table on top.