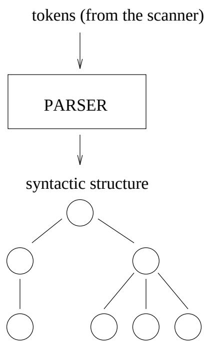
The Scanner and the Parser

- What does the scanner (lexical analyser) do?
- What other kinds of analysis needs to be done?
- What parts of a compiler do these other kinds of analysis?

The Parser



Tasks Performed by the Parser

- Recognising syntactic structure.
- Verifying correct syntax.
- Handling errors.
- Building the syntax tree.

Defining the syntax of a programming language

The parser is driven by a formal description of the syntax of the language it parses.

Regular expressions are not sufficient to describe the structures found in programming languages.

For example, consider

```
for ... loop
    ...
    for ... loop
        for ... loop
           ...
        endloop
    endloop
    ...
endloop
```

Regular expressions cannot define nested constructs.

Context Free Grammars

Context free grammars provide a way of specifying the syntactic structure of modern programming languages.

A context free grammar contains rules.

```
<expr> ::= <expr> + <term> | <term>
<term> ::= <term> + <factor> | <factor>
<factor> ::= id | number | (<expr>)
```

This notation is called Backus-Naur Form (BNF)

Context Free Grammars

The grammar rules contain terminal and nonterminal symbols. The terminal symbols are the symbols of the language being parsed, while the nonterminal symbols are used in the grammar to represent whole phrases of the language.

Each rule has a left side consisting of a single nonterminal symbol. (This is what is meant by "context free").

Each rule has one or more alternative right sides; each right side consists of a string of terminal and nonterminal symbols.

One of the nonterminal symbols is defined to be the start symbol; in the example, the start symbol is $< expr >$.

Alternative forms of BNF

BNF can be simplified in various ways.

```
PROGRAM -> SEQU
SEQU    -> COMMAND | COMMAND ; SEQU
COMMAND -> id := EXPR |
   if EXPR then SEQU else SEQU endif |
   while EXPR do SEQU endwhile
EXPR    -> number | id
```

```
PROGRAM -> SEQU
SEQU    -> COMMAND [; COMMAND]*
COMMAND -> id := EXPR | ... etc.
```

Context Free Grammars

A context free grammar (CFG) consists of:

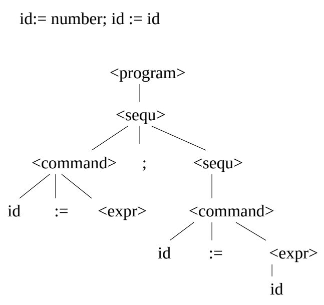
- a finite terminal vocabulary, T,
- a finite nonterminal vocabulary, N where $N \cap T = \{\}$.
- a start symbol, $S \epsilon N$, and
- a finite set of productions, P, of the form $A \rightarrow X_1 \ldots X_m$, where $A \epsilon N$ and $X_i \epsilon N \cup T for 1 \leq i \leq m$

We often write $G = (T, N, S, P)$ to describe a grammar.

The set $V = N \cup T$ is called the vocabulary of a grammar.

Parse Trees

The derivation of a string from a grammar can be represented by a parse tree.



Approaches to parsing

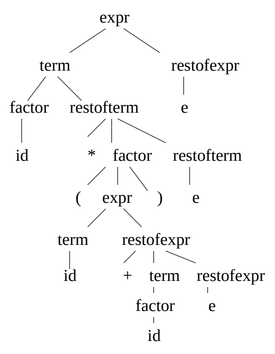
- Top down parsing – start with start symbol and build tree from the root down
- Bottom up parsing – start with input tokens and build tree from the leaves up

Parse Trees and Abstract Syntax Trees

Consider the grammar

```
expr ::= term restofexpr
restofexpr ::= + term restofexpr | e
term ::= factor restofterm
restofterm ::= * factor restofterm | e
factor ::= number | id | (expr)
```

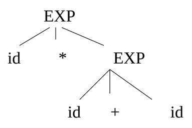
A Parse Tree for id * (id + id)



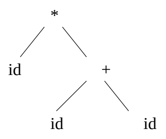
This parse tree is cumbersome, and not necessarily well suited to the later stages of compilation.

Abstract Syntax Trees for id * (id + id)

An abstract syntax tree for id * (id + id) could look like this



or like this



Abstract syntax trees are compact, and adapted to the needs of the later stages of compilation.

Different Parsing Methods

- top-down parsing with backtracking
- top-down deterministic parsing
  – e.g. LL(K), LL(1)
- bottom-up deterministic parsing
  – e.g. LR(K), LALR(K), LALR(1)
- operator precedence parsing,
  a bottom up parsing method that is especially useful for expressions like $a + b * c$

Usually, the grammar has to be restricted in some way to make each of these parsing methods work.

For example, top down parsing methods cannot deal with left-recursive grammars:

```
    E ::= E + T
    T ::= T * F | F
    F ::= (E) | id
```

Instead, the grammar should look like this for top-down parsing:

```
    E  ::= T E'          E' ::= + T E' | e
    T  ::= F T'          T' ::= * F T' | e
    F  ::= (E) | id
```

where e stands for the empty string.

Recursive Descent Parsing

Recursive descent parsing is very easy to implement.

For each nonterminal, a routine is written to parse all the strings that can be derived from the nonterminal. This means that the program corresponds closely to the grammar.

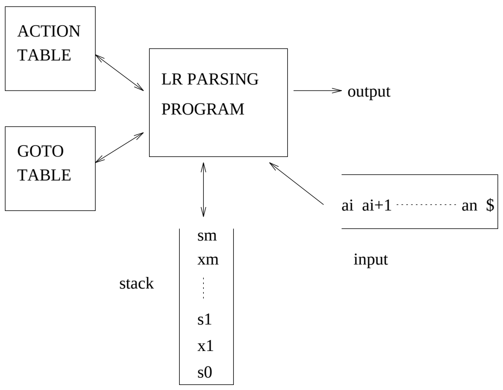
For example, given the rule

```
    expr -> term restofexpr
```

our pseudocode might be

```
    parse_expr = parse_term;
                 parse_restofexpr;
                 return tree
```

LR(1) Parsing



a1, . . . , an are terminal symbols
$ is an end of file marker
x1, . . . , xm are grammar symbols (terminals or nonterminals or both)
s0, . . . , sm are state symbols

How the LR(1) Parser Works

1. It determines *sm*, the state symbol on top of the stack, and *ai*, the current input symbol.

2. It then consults the action table entry for *sm* and *ai*,

   action[sm,ai]

   which has one of the values

   shift s, reduce A ::= b, accept, or error.

3. Then it carries out the prescribed action, after which it will repeat the whole process, or terminate.

The LR Parsing Algorithm

```
a:= scan (input) /* first token */
REPEAT FOREVER
  s := state on top of stack
  IF action[s,a] = shift s'
     THEN push a onto the stack
          push s' onto the stack
          a := scan(input) /* next token */
  ELSEIF action[s,a] = reduce A ::= b
     THEN pop (2 * |b|) symbols off stack
          s' := new state on top of stack
          push A onto the stack
          push goto[s',A] onto the stack
          output rule A ::= b
  ELSEIF action[s,a] = accept
     THEN exit successfully
  ELSE /* action[s,a] = error */
     execute error routine
END REPEAT
```

Example

Consider the expression grammar

```
Rule 1:     E ::= E + T
Rule 2:     E ::= T
Rule 3:     T ::= T * F
Rule 4:     T ::= F
Rule 5:     F ::= (E)
Rule 6:     F ::= id
```

Suppose we want to parse the expression

```
id * id + id
```

Parsing Action Table

	Input Token					
State	id	+	*	(	)	$
0	s5			s4		
1		s6				accept
2		r2	s7		r2	r2
3		r4	r4		r4	r4
4	s5			s4		
5		r6	r6		r6	r6
6	s5			s4		
7	s5			s4		
8		s6			s11	
9		r1	s7		r1	r1
10		r3	r3		r3	r3
11		r5	r5		r5	r5

si – shift and stack state i

rn – reduce using rule number n

accept – accept

blank – error

Goto Table

	Nonterminal		
State	E	T	F
0	1	2	3
1			
2			
3			
4	8	2	3
5			
6		9	3
7			10
8			
9			
10			
11			

Parsing $id * id + id$

Stack	Input	Action
0	id*id+id$	shift 5
0id5	*id+id$	reduce F ::= id
0F3	*id+id$	reduce T ::= F
0T2	*id+id$	shift 7
0T2*7	id+id$	shift 5
0T2*7id5	+id$	reduce F ::= id
0T2*7F10	+id$	reduce T ::= T*F
0T2	+id$	reduce E ::= T
0E1	+id$	shift 6
0E1+6	id$	shift 5
0E1+6id5	$	reduce F ::= id
0E1+6F3	$	reduce T ::= F
0E16+T9	$	reduce E ::= E+T
0E1	$	accept

Shift-Reduce and Reduce-Reduce Conflicts

Suppose a grammar included a rule like

```
T ::= X Y | X.
```

Having read an X, the parser could reduce it to a T, or could shift more symbols that might be a Y. This is called a *shift-reduce conflict*.

Another kind of conflict is know as a *reduce-reduce conflict*. This occurs when the parser cannot decide which of several possible productions to apply in reducing.

If it is impossible to generate LR(1) action and goto tables for a grammar, then the grammar is said to be not LR(1). Fortunately, most programming languages can be defined using LR(1) grammars.

+ +

LALR Parsing

A grammar for a typical programming language might have 50 to 100 terminals and about 100 productions (grammar rules). The LR(1) tables for such a language would have thousands of states!

In practice, a technique called LALR is used, giving rise to several hundred states for a language like Pascal.

LALR(1) is slightly more restrictive than LR(1), but experience has shown it to be adequate for the kind of languages we want to parse.

yacc is an LALR parser generator which was written by S.C. Johnson in the early 1970s.

+ +

Building the Syntax tree

The parser builds the abstract syntax tree by invoking a tree building routine each time a grammar rule is applied.

The tree building procedures are defined using an extended grammar, called an *attribute grammar*.

+ +

Building the Syntax tree

For example, if we were using an LR parser to parse expressions, we might use an attribute grammar like this to build the syntax tree:

```
E ::= E1 + T {E.treeptr :=
               mknode(E1.treeptr,'+',T.treeptr)}
E ::= T      {E.treeptr := T.treeptr}
T ::= T1 * F {T.treeptr :=
               mknode(T1.treeptr,'*',F.treeptr)}
T ::= F      {T.treeptr := F.treeptr}
F ::= (E)    {F.treeptr := E.treeprt}
F ::= id     {F.treeptr := mkleaf('id')}
```

where mknode returns an internal node of the syntax tree, and mkleaf returns a leaf node.

+ +

Building the Syntax tree

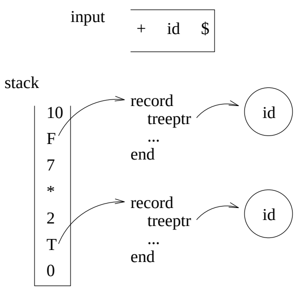
This attribute grammar deals with a single attribute, treeptr, a pointer to a fragment of the syntax tree.

treeptr is an attribute of E, T and F.

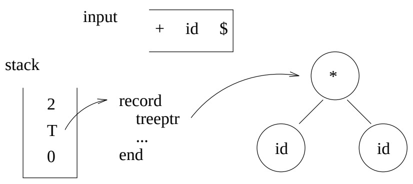
treeptr is called a *structural attribute*, because it contains information about the structure of the abstract syntax tree.

During a bottom up parse of $id * id + id$, using the parsing action and goto tables shown previously, the stack, input, and syntax tree would appear as illustrated.

Building the Syntax tree



ACTION: reduce T::= T1 * F {T.treeptr := mknode T1.treeptr, *, F.treeptr}
GOTO(0 , T) = 2


input
+ id $
stack
2
T
0
record
treeptr
...
end
*
id
id