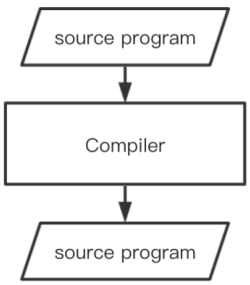
笔记

Principles of Compilers

1. Compiler

First of all, what is the compilers? Nowadays, programming generally uses high-level languages, such as C or Java. However the computer only understands its own instructio n system and can only run the programs written in machine language. Therefore, between these two languages, a translator is needed -- **the compiler is this translator**. So, simply stated, a compiler is a program that can read a program in one language -- the **source language** -- like I mentioned C or Java, and translate it into an equivalent program in another language -- the **target language**, such as the mechine language.

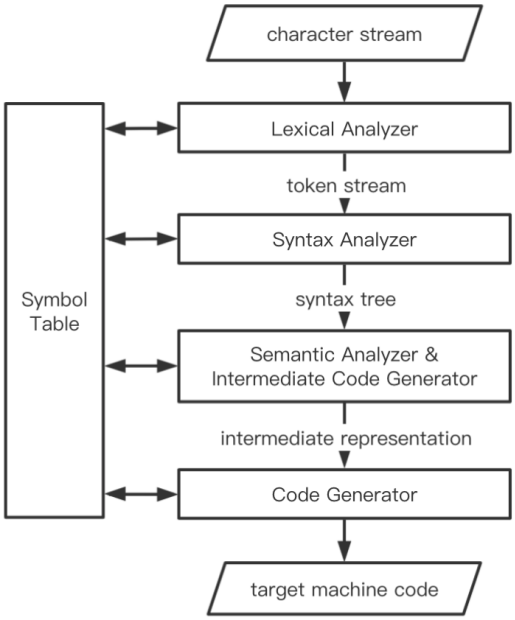


A compiler

Now we know what the compiler does, translation, next step we need to know how it does.

1.1 The Structure of a Compiler

The three main steps in the compilation process, are lexical analysis, syntax analysis and sematic analysis. We can first simply understand these three steps as analyzing words, grammar, and semantics. Now I'll talk about them one by one.



Phases of a compiler

2. Lexical Analysis

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis.

- Tokens

A token is a pair consisting of a token name and an optional attribute value.
<token–name, attribute–value>

- Patterns

A pattern is a description of the form that the lexemes of a token may take.

- Lexemes

A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Example:

A source program contains the assignment statement:
position = initial + rate * 60
shows the representation of the assignment statement after lexical analysis as tokens:
<id, 1> <=> <id, 2> <+> <id, 3> <*> <number, 60>

2.1 Regular Expressions

2.1.1 Strings and Languages

- The empty string, denoted ε */epsilon/*, is the string of length zero.
- An **alphabet** is any finite set of symbols.
- A **language** is any countable set of strings over some fixed alphabet
- Terms for Parts of Strings: **prefix**, **suffix**, **substring**, **proper**(prefix, suffix, substring), **subsequence**.

2.1.2 Operations on Languages

Operation	Definition and Notation
Union of L and M	$L \cup M =$ {s | s is in L or s is in M}
Concatenation of L and M	$LM =$ {st | s is in L and t is in M}
Kleene closure of L	$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots L^n \ldots = \bigcup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = L^1 \cup L^2 \cup \ldots L^n \ldots = \bigcup_{i=1}^{\infty} L^i$

2.1.3 Regular Expressions

Example:

$letter(letter|digit)^*$

2.1.4 Regular Definitions

- Form:

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

...

$d_n \rightarrow r_n$

  - d_i is a new symbol
  - r_i is regular expression

- Example: The regular definition of unsigned number:

$digit \rightarrow 0|1|2\ldots|9$

$digits \rightarrow digit^+$

$number \rightarrow digits(.digits)?$

$letter \rightarrow [A-Za-z]$

$id \rightarrow letter(letter|digit)^*$

$if \rightarrow if$

$else \rightarrow else$

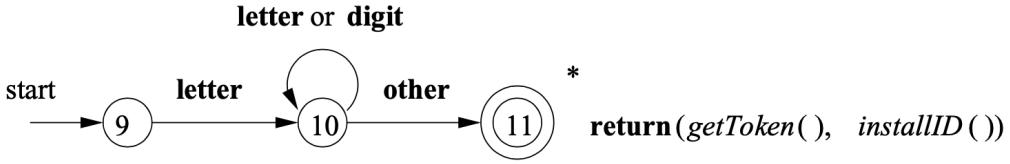
$relop \rightarrow \ < \ | <= | > | >= | =$

2.2 NFA & DFA

2.2.1 Transition Diagrams

As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called '**transition diagrams**'. Transition diagrams have a collection of nodes or circles, called **states**. **Edges** are directed from one state of the transition diagram to another. Each edge is labeled by a symbol or set of symbols.
Some important conventions:

1. Certain states are said to be **accepting** or **final**.
2. ***** means retracting the forward pointer one position.
3. One state is designated the **start** state, or initial state.

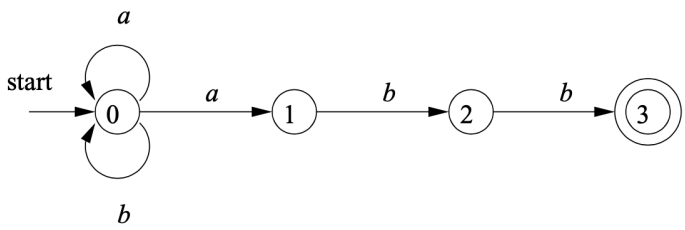


A transition diagram for id's and keywords

2.2.2 NFA

NFA (Nondeterministic Finite Automata) have restrictions on the labels of their edges. A symbol can label several edges out of the same state, and ε, the empty string is a possible label.

Transition graph:



Transition table:

STATE	a	b	ε
0	{0, 1}	{0}	Ø
1	Ø	{2}	Ø
2	Ø	{3}	Ø
3	Ø	Ø	Ø

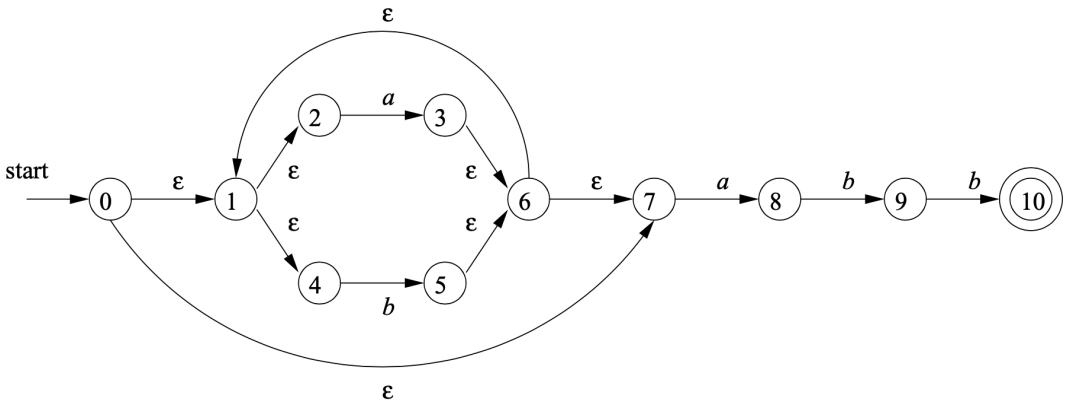
2.2.3 DFA

DFA (Deterministic Finite Automata) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

2.2.4 Conversion of an NFA to a DFA

For example: $(a|b)^*aab$

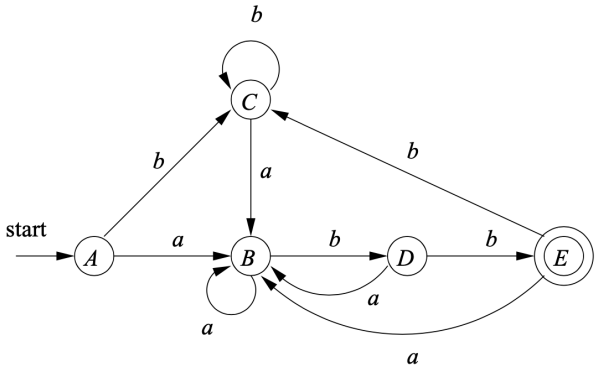
1. Transition graph of NFA:



2. Transition table:

NFA STATE	DFA STATE	a	b
$\{0,1,2,4,7\}$	A	B	C
$\{1,2,3,4,6,7,8\}$	B	B	D
$\{1,2,4,5,6,7\}$	C	B	C
$\{1,2,4,5,6,7,9\}$	D	B	E
$\{1,2,4,5,6,7,10\}$	E	B	C

3. Transition graph of DFA:



3. Syntax Analysis

The second phase of the compiler is syntax analysis or parsing. The parser obtains a string of tokens from the lexical analyzer and output a parse tree. The methods commonly used in parser can be classified as being either top–down or bottom–up.

3.1 Context–Free Grammars

A context–free grammar (grammar for short) consists of :

- Terminals
- Nonterminals
- A start symbol
- Productions. Each production consists of:
  - A nonterminal called the head or left side of the production
  - The symbol –>
  - A body or right side consisting of zero or more terminals and nonterminals

A formal grammar is considered ”context free” when its production rules can be applied regardless of the context of a nonterminal. It does not matter which symbols the nonterminal is surrounded by, the single nonterminal on the left hand side can always be replaced by the right hand side.

Non–Context–Free Language

The production likes: $\alpha E \beta \rightarrow \alpha \gamma \beta$

3.1.1 Derivations

For example:

- E => –E: E derives –E
- $\alpha A \beta \Rightarrow \alpha \gamma \beta$.: derives in one step
- $\alpha \overset{*}{\Rightarrow} \gamma$: derives in zero or more steps
- $\overset{+}{\Rightarrow}$: derives in one or more steps

Leftmost derivations: the leftmost nonterminal in each sentential is always chosen.

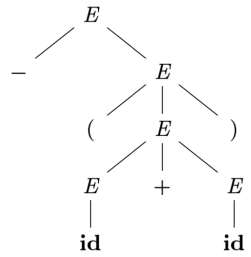
Rightmost derivations: the rightmost nonterminal in each sentential is always chosen.

3.1.2 Parse Tree

A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals.

For example: E –> E+E | E*E | –E | (E) | id

Parse tree for –(id+id)



3.1.3 Eliminating Ambiguity

For example: $E \to E + E \mid E * E \mid (E) \mid id$

result:

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid id$

3.1.4 Elimination of Left Recursion

For example: $E \to Ea \mid b$

result:

$E \to bE^{'}$

$E^{'} \to aE^{'} \mid \varepsilon$

3.1.5 Left Factoring

For example: $E \to \alpha\beta_1 \mid \alpha\beta_2$

result:

$E \to \alpha E^{'}$

$E^{'} \to \beta_1 \mid \beta_2$

3.2 Top–Down Parsing

3.2.1 LL(1) Grammar

The first 'L' in LL(1) stands for scanning the input from left to right, the second 'L' for producing a leftmost derivation, and the '1' for using one input symbol of lookahead at each step to make parsing action decisions.

3.2.2 FIRST and FOLLOW

The construction of both top–down and bottom–up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G.

For example:

$S \to Ade \mid A$

$A \to ab \mid Bc \mid \varepsilon$

$B \to b$

- FIRST

FIRST(A), where A is any string of grammar symbols, to be the set of terminals that begin strings derived from A. In example, FIRST(A) = {a, b, ε}

- FOLLOW
  FOLLOW(A), for nonterminal A, to be the set of terminals that can appear immediately to the right of A. In example, FOLLOW(A) = {d, ε}

3.2.3 Nonrecursive Predictive Parsing

Nonrecursive predictive parsing is table-driven . The parser mimics a leftmost derivation.
For example:

- There is a grammar:
  $G(E) = (\{E, F, T\}, \{id, num, =, +\}, P, E)$
  $P: \quad E \to D = F$
  $F \to T + T$
  $T \to id \mid num$
  $D \to id$
- String:
  id=id+num
- FIRST and FOLLOW:

	FIRST	FOLLOW
E	{id}	{$}
F	{id, num}	{$}
T	{id, num}	{$}
D	{id}	{=}

- Parsing table:

	id	num	=	+	$
E	E->D+F				
F	F->T+T	F->T+T			
T	T->id	T->num			
D	D->id				

- Derivation process：

MATCHED	STACK	INPUT	ACTION
	E$	id=id+num$	
	D=F$	id=id+num$	E->D=F
	id=F$	id=id+num$	D->id
id	=F$	=id+num$	match *id*
id=	F$	id+num$	match =
id=	T+T$	id+num$	F->T+T
id=	id+T$	id+num$	T->id
id=id	+T$	+num$	match *id*
id=id+	T$	num$	match +
id=id+	num$	num$	T->num
id=id+num	$	$	match *num*

3.3 Bottom-Up Parsing

A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

3.3.1 Reductions

A reduction is the reverse of a step in a derivation.

We can think of bottom–up parsing as the process of “reducing” a string to the start symbol of the grammar. At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of production.
The key decisions during bottom–up parsing are about when to reduce and about what production to apply, as the parse proceeds.
For example: id+num, T+num, T+T, F

3.3.2 Handle

Informally, a ’handle’ is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.
For example:

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$

A rightmost derivation in reverse can be obtained by ”handle pruning”.

3.3.3 Shift–Reduce Parsing

For example:

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \to \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \to F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift
$\$ T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$ T * F$	$\$$	reduce by $T \to T * F$
$\$ T$	$\$$	reduce by $E \to T$
$\$ E$	$\$$	accept

3.3.4 LR Parsing

The most prevalent type of bottom–up parser today is based on a concept called LR(k) parsing. The ’L’ is for left–to–right scanning, the ’R’ for constructing a rightmost derivation in reverse, and the k for number of input symbols of lookahead that are used in making paring decisions. LR parsers are table–driven.
The principal drawback of the LR method is that it is too much work to construct an LR parser by hand for a typical programming–language grammar.
For example:
Parse input *id=id+num* by LR Parsing:

- Grammar

  $G(E) = (\{E, F, T\}, \{id, num, =, +\}, P, E)$

  $P: \quad E \to D = F$

  $F \to T + T$

  $T \to id \mid num$

  $D \to id$

- Construct new grammar G'

  1) $E \to D = F$

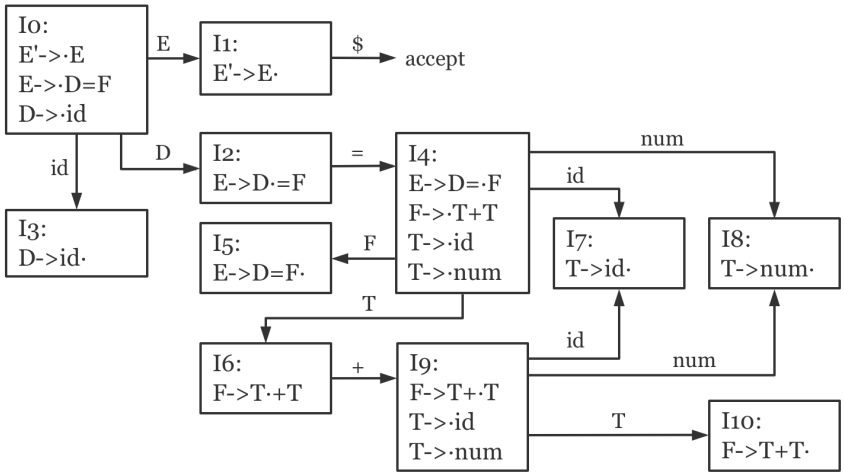
2) $F \rightarrow T + T$

3) $T \rightarrow id$

4) $T \rightarrow num$

5) $D \rightarrow id$

- Automaton for the expression grammar



- LR Parsing Table

  The parsing table consists of two parts: a parsing–action function ACTION and a goto function GOTO.

STATE	ACTION					GOTO			
	id	num	=	+	$	E	D	F	T
0	s3					1	2		
1					acc				
2				s4					
3				r6					
4	s7	s8						5	6
5					r2				
6				s9					
7				r4	r4				
8				r5	r5				
9	s7	s8							10
10					r3				

- Reduction Process

	STACK	SYMBOLS	INPUT	ACTION
1)	0		id=id+num$	s3
2)	03	id	=id+num$	r6
3)	02	D	=id+num$	s4
4)	024	D=	id+num$	s7
5)	0247	D=id	+num$	r4
6)	0246	D=T	+num$	s9
7)	02469	D=T+	num$	s8
8)	024698	D=T+num	$	r5
9)	0246910	D=T+T	$	r3
10)	0245	D=F	$	r2
11)	01	E	$	acc

4. Semantic Analysis

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during code generation.

4.1 Token

The tokens is output of lexical analyzer. It is a pair consisting of a token name and an optional attribute value.
<token–name, attribute–value>

4.2 Symbol Table

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used).
The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.

1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE

4.3 Attribute Grammar

In the program, semantic analysis and syntax analysis are carried out synchronously, each production corresponds to a semantic analysis subprogram, and the semantic subprogram is executed when the syntax analyzer completes a derivation or reduction.

Production	Semantic rules
$E \to E * E$	$E.val = E^{(1)}.val * E^{(2)}.val$
$E \to E + E$	$E.val = E^{(1)}.val + E^{(2)}.val$
$E \to (E)$	$E.val = E^{(1)}.val$
$E \to id$	$E.val = id.Lexval$
$E \to num$	$E.val = num.Lexval$