COMPILER

SCANNER

* AKA tokenizer, lexical analyzer
* Reads characters from source code, and converts them into tokens (aka lexemes)

Token example

**Input =>** “c = a + b;”

|  |  |
| --- | --- |
| **Token** | **Type** |
| c | Identifier |
| = | Assignment operator |
| a | Identifier |
| + | Arithmetic operator |
| b | Identifier |
| ; | Separator |

**TEST**

Build a symbol table from the below input:

int a, b, c;

float d, e;

a = b = 5;

c =6;

if (a > b) {

c = a – b;

e = d – 2.0;

} else {

d = e + 6.0;

b = a + c;

}

Desired output

Identifiers: int, float, if, else

math operators: +, -, =

logical operators: >

numerical values: 5, 6, 2.0, 6.0

other: (, ), {, }, ;

SYMBOL TABLES

* A data structure, most likely a dictionary which stores identifier information, such as their values, name, attributes, etc.
* The lexer initially (generally), stores the identifiers in the symbol table, and is accessed and update by other parts of the compiler.

Consider the string: “var age = 10;”.

**Symbol\_table** = [

{

id: 0,

name: “age”,

value: null // may be populated now or later in the parse / code gen phase

}

]

CONTEXT FREE GRAMMAR (CFG)

* A CFG is a set of recursive rules used to generate patterns of strings.

Example

S -> AB // S is common for representing the start symbol

A -> aAA | aA | a

B -> bB | b

Non terminals = {S, A, B}

Terminals = {a, b}

* It is common to represent non-terminals in upper case.

Example of a CFG that generates arithmetic expressions

1. <expression> -> number
2. <expression> -> (<expression>)
3. <expression> -> <expression> + <expression>
4. <expression> -> <expression> – <expression>
5. <expression> -> <expression> \* <expression>
6. <expression> -> <expression> / <expression>

Using the example above, the steps of deriving the following expression following are:

*(4 + 5) \* (2 – 6)*

*<expression> -> 4 (rule 1)*

*<expression> -> 5 (rule 1)*

*<expression> -> 4 + 5 (rule 3)*

*<expression> -> (4 + 5) (rule 2)*

*<expression> -> 2 (rule 1)*

*<expression> -> 6 (rule 1)*

*<expression> -> 2 – 6 (rule 4)*

*<expression> -> (2 – 6) (rule 2)*

*<expression> -> (4 + 5) \* (2 – 6) (rule 5)*

*Note: there are many ways to do this ^^*

Parse trees

* CFGs can be modelled as *parse trees*.
* The nodes of the tree represent the symbols.
* The edges represent the use of production rules.
* The leaves represent the terminals.

*Example:*

Generate the string *“a + a – a”*

A

a

-

+

a

a

A

A

A

A

A -> A + A | A – A | a

Production rules for basic English

S -> NounPhrase VerbPhrase

NounPhrase -> Adj NounPhrase | Adj Noun

VerbPhrase -> Verb NounPhrase

Adj -> the

Noun -> monkey | banana

Verb -> ate

From the above production rules; one could make the sentence:

*The monkey ate the banana*

But **not** make:

*Monkey the ate*

**TEST**

* Go through the steps for the 2 sentences above using the above grammar, and visually see how the sentences are made and why one is valid and the other is not.

PARSER

* AKA syntax analyser.
* Analyses strings and checks whether the string conforms to a particular grammar.
* Usually generates a data structure in the form of a parse tree or abstract syntax tree (AST), which is used by the *code generator* to create the target code output.

Recursive descent parsing

A top-down parsing technique that constructs the parse tree from the top, and the input is read from left to right.

*Example:*

Given the grammer:

A -> abC | aBd | aAD

B -> bB | Ɛ // Ɛ = empty string

C -> d | Ɛ

D -> a | b | c

Is this input valid: “a a b a”

// **bold** represents pointers

// when you encounter an empty string, increment the production rule pointer

Step 1

“**a** a b a”

// using production rule 1

b

**a**

A

C

// there is a match – increment both pointers…

Step 2

“a a **b** a”

// there is no match. Production rule 1 is expecting a “b”

// both pointers now need to go back to the start as production rule 1 does not

work with our input string.

Step 3

Using production rule 2, our first character matches, but our second does not!

A

B

d

**b**

a

B

// no match again!

Step n

The input string is valid, using the aAD production rule.

**TEST –** sketch the parse trees for each stage of this rule to visually see how this string is valid.

ABSTRACT SYNTAX TREE

* A tree representation of the abstract syntactic structure of source code.
* Used to generate the target code.
* Visit astexplorer.net to get an idea of how it looks.

|  |  |
| --- | --- |
| Declaration | Specifies properties of an identifier; e.g.  const name; |
| Expression | Any legal combination of symbols that represents a value; e.g.  num \* 10; |
| Statement | A statement ‘does something’; e.g.  Assign a value: const x = 10;  printf(“hello world”); |
| Identifier | The name given to variables, functions, classes, etc. e.g.  var **name** = ‘ali’;  class **Printer** {} |
| Literal | The value of a variable |
| Callee | Function being called |

* As tokens are being parsed and checked that they match the grammar, a syntax tree is generally built up, which looks something like:

*const age = 29; // start index 0, end index 14*

*>> output*

*Node [*

*Start: 0,*

*End: 14,*

*Type: ‘Program’,*

*Body: [*

*Start: 0,*

*End: 4,*

Kind: ‘Const’,

Type: ‘VariableDeclaration’,

Declarations: [

{

Start: 6,

End: 13,

Type: ‘Identifier’,

Id: {

Type: ‘Identifier’,

Start: 6,

End: 10,

Name: ‘age’

},

Init: {

Start: 12,

End: 14,

Type: ‘Literal’,

Raw: ‘28’

}

}

]

*]*

*]*

The above will be built up most-likely using recursion:

const ast = {

start: 0,

end: 14,

body: [],

type: ‘Program’

}

*While(this.nextToken() !== types.eof) {*

*const statement = this.parseStatement();*

*ast.body.push(statement);*

*}*

*return node;*

CODE GENERATION

* Translates the parse tree into the target language.