CPS2000
Compiler Theory & Practice

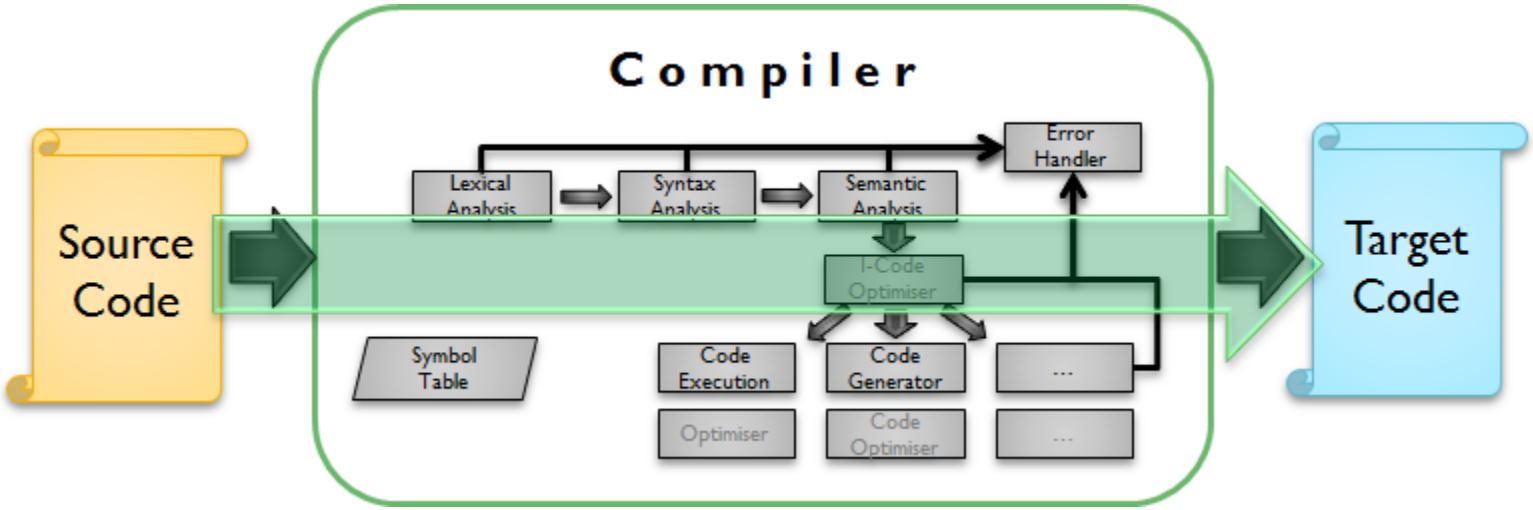
Notes on Handcrafting a Parser

Gordon Mangion

What is a Compiler?

What is a Compiler?

- Is essentially a complex **function** which **maps** a program in a **source** language onto a program in the **target** language.



Translation

Source Code

```
function Sqr( x : int ) : int
{
    let n : int = x;
    n <- n * n;
    return n;
}
```

Target Code

```
push        ebp
mov         ebp,esp
sub         esp,0CCh
push        ebx
push        esi
push        edi
lea         edi,[ebp-0CCh]
mov         ecx,33h
mov         eax,0CCCCCCCCh
rep stos    dword ptr es:[edi]
mov         eax,dword ptr [x]
mov         dword ptr [n],eax
mov         eax,dword ptr [n]
imul        eax,dword ptr [n]
mov         dword ptr [n],eax
mov         eax,dword ptr [n]
pop         edi
pop         esi
pop         ebx
mov         esp,ebp
pop         ebp
ret
```

Translation

Source Code

```
function Sqr( x : int ) : int
{
    let n : int = x;
    n <- n * n;
    return n;
}
```

Target Code

```
push        ebp
mov         ebp,esp
sub         esp,0CCh
push        ebx
push        esi
push        edi
lea         edi,[ebp-0CCh]
mov         ecx,33h
mov         eax,0CCCCCCCCh
rep stos    dword ptr es:[edi]

mov             eax,dword ptr [x]
mov             dword ptr [n],eax

mov             eax,dword ptr [n]
imul            eax,dword ptr [n]
mov             dword ptr [n],eax

mov             eax,dword ptr [n]

pop         edi
pop         esi
pop         ebx
mov         esp,ebp
pop         ebp
ret
```

Production Rules

Context Free Grammar

A → B ‘and’ C

B → ‘true’ B → ‘false’

C → B

Terminals: ‘and’, ‘true’, ‘false’

Non-Terminals: A, B, C

Patterns: Think of identifiers

BNF - Backus Naur Form

Context Free Grammar

A ::= B ‘and’ C

B ::= ‘true’ | ‘false’

C ::= B

Terminals: ‘and’, ‘true’, ‘false’

Non-Terminals: A, B, C

Patterns: Think of identifiers

Grammar EBNF

Grouping

( )

Repetitions

[ ] - 0 or 1 time

{ } - 0 or more times

Ranges []

<Letter> ::= [A-Za-z]

<Digit> ::= [0-9]

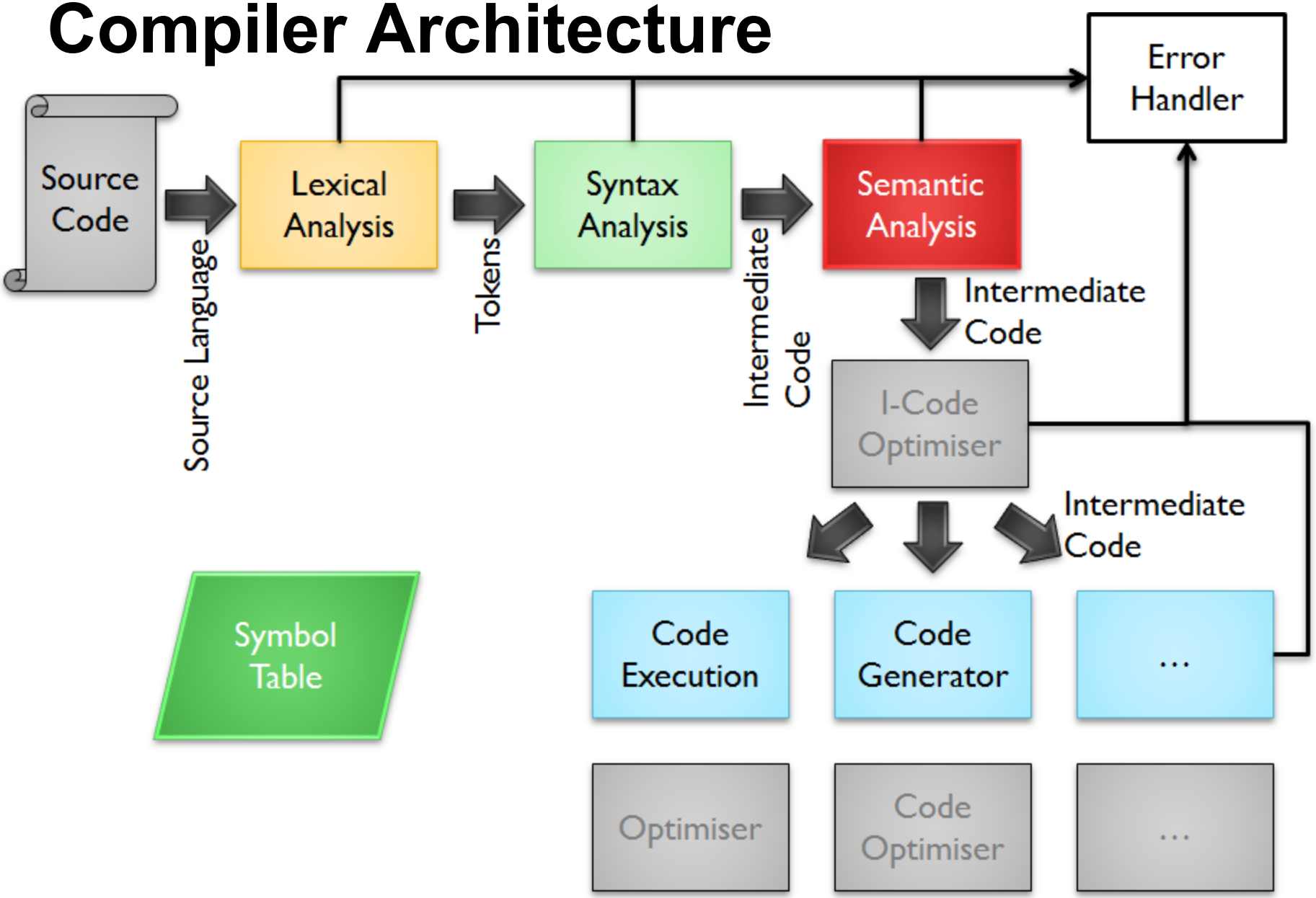
Grammar Example

<Letter> ::= [A-Za-z]
<Digit> ::= [0-9]

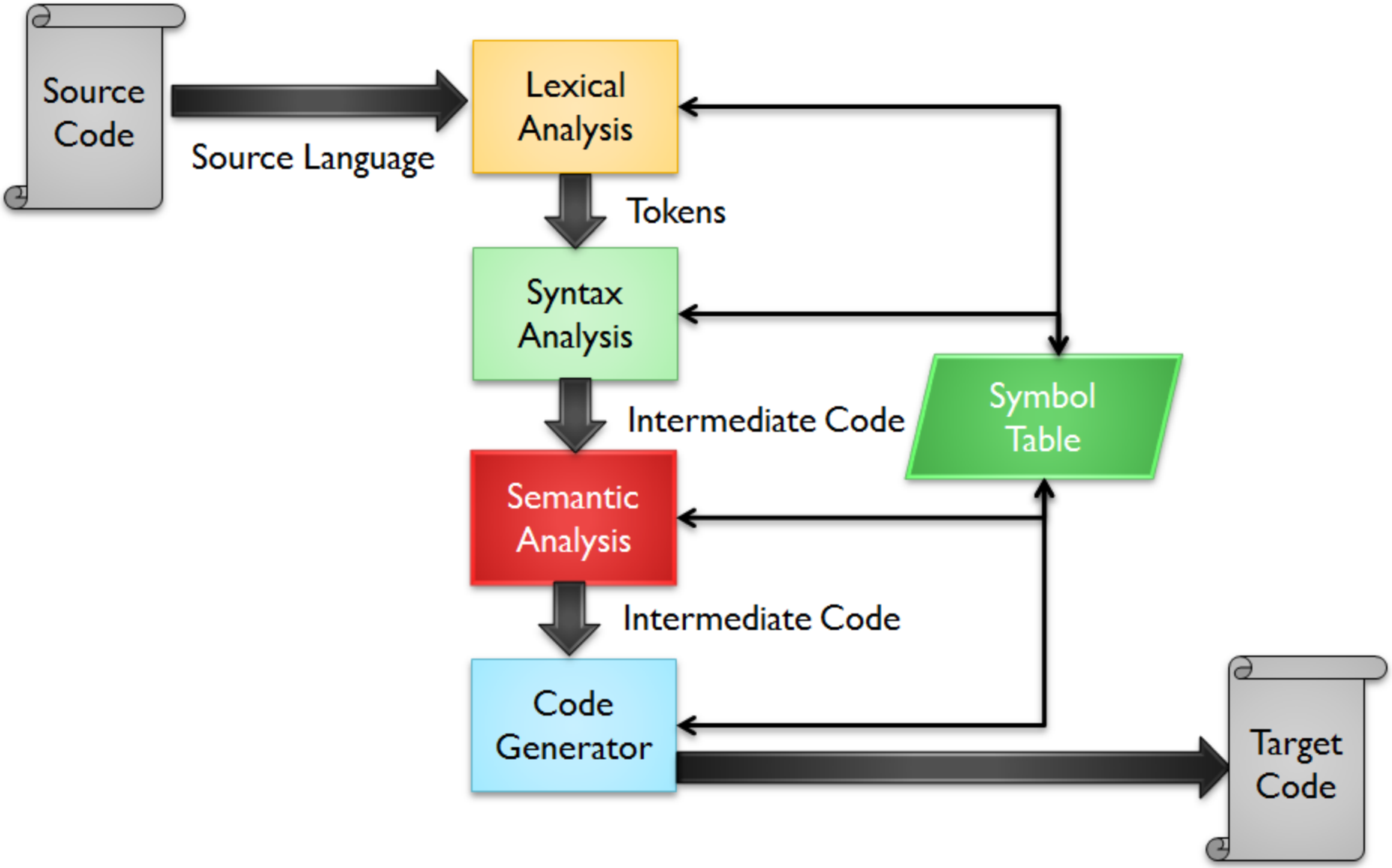
<IntegerLiteral> ::= <Digit> { <Digit> }
<Identifier> ::= (<Letter> | ‘_’) { <Letter> | “_” | <Digit> }

<Expression> ::= <Identifier> | <IntegerLiteral>
<Assignment> ::= ‘**set**’ <Identifier> ‘**<-**’ <Expression>
<Statement> ::= <Assignment> ‘;’ | <Expression> ‘;’
<Program> ::= { <Statement> }

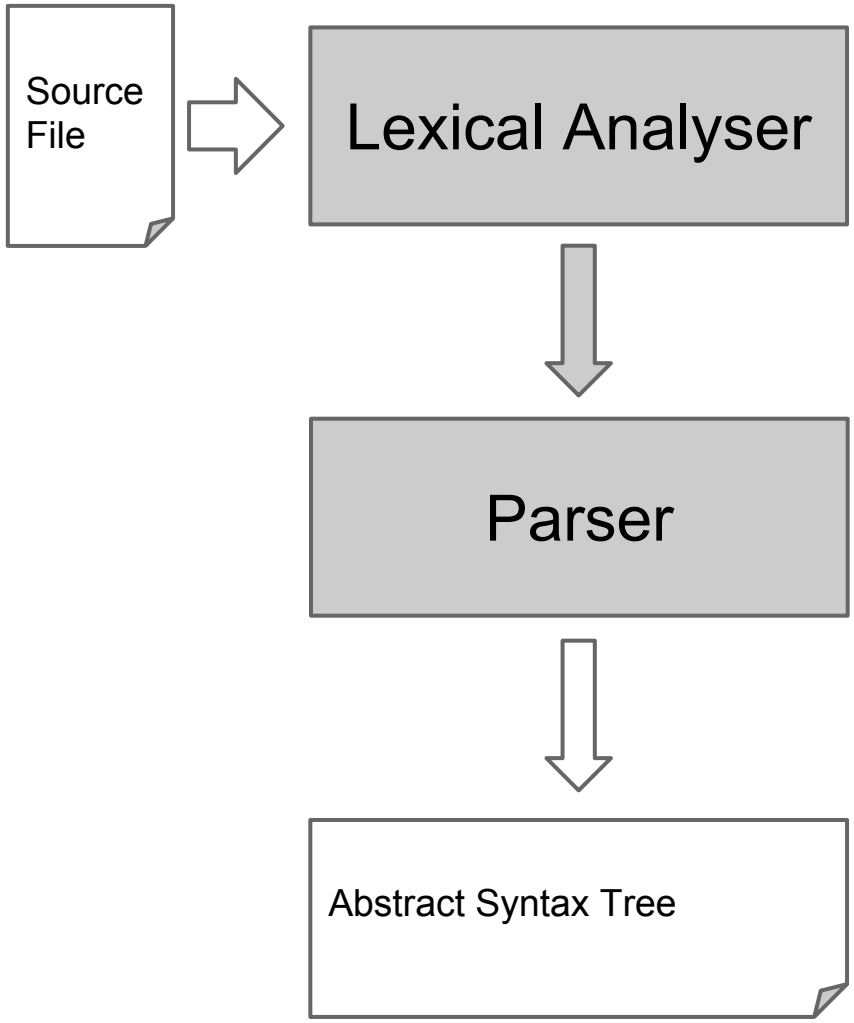
Compiler Architecture



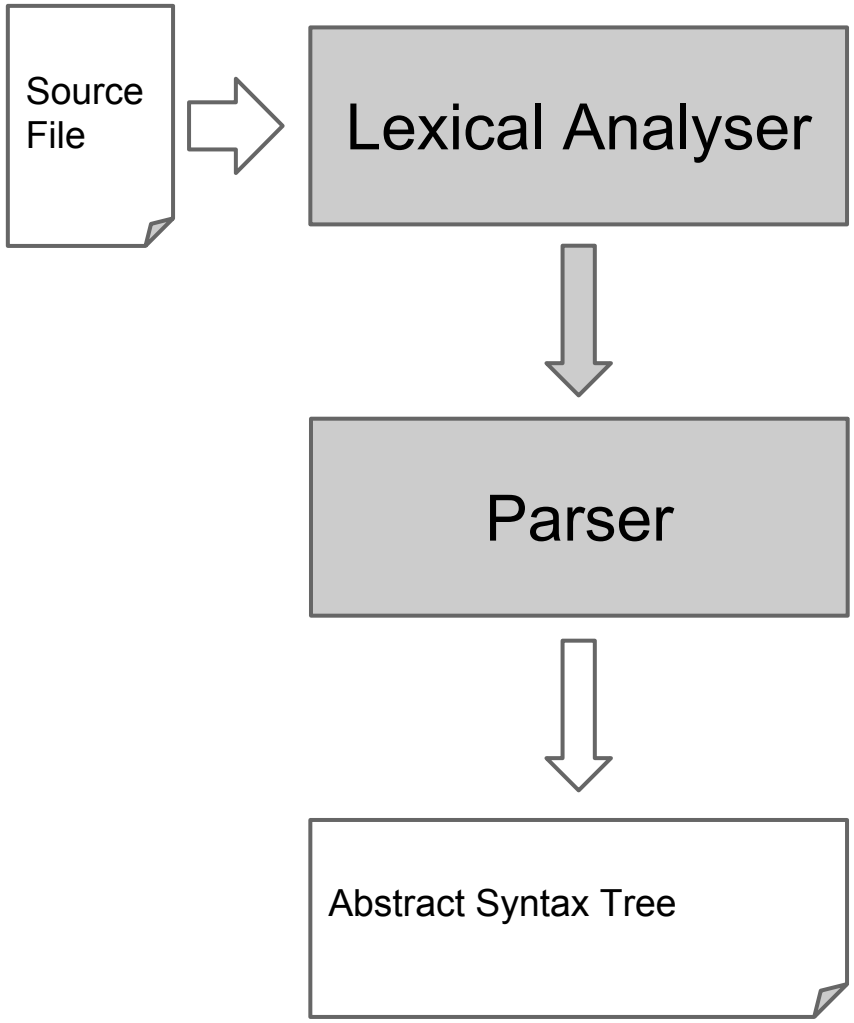
Simplified Compiler Architecture



Compiler



Compiler



Keyword Table

Symbol Table

Abstract Syntax Tree

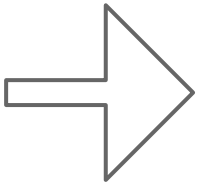
Error Module?

Notice

As always, the Implementation can be done in many many ways.

Lexical Analysis

1. Read from source file
   - Position Accounting
2. Categorise the lexeme
3. Output Tokens



- Whitespace
- Literals
  - Integers / Real ...
  - Strings
  - Characters
- Comments
- Operators
- Keywords
- Identifiers

Lexical Analysis - Token

TokenKind:

- tk_eof
- tk_if
- tk_true
- tk_plus
- tk_identifier

TokenValue:

- Value ← Lexeme

Position Information?

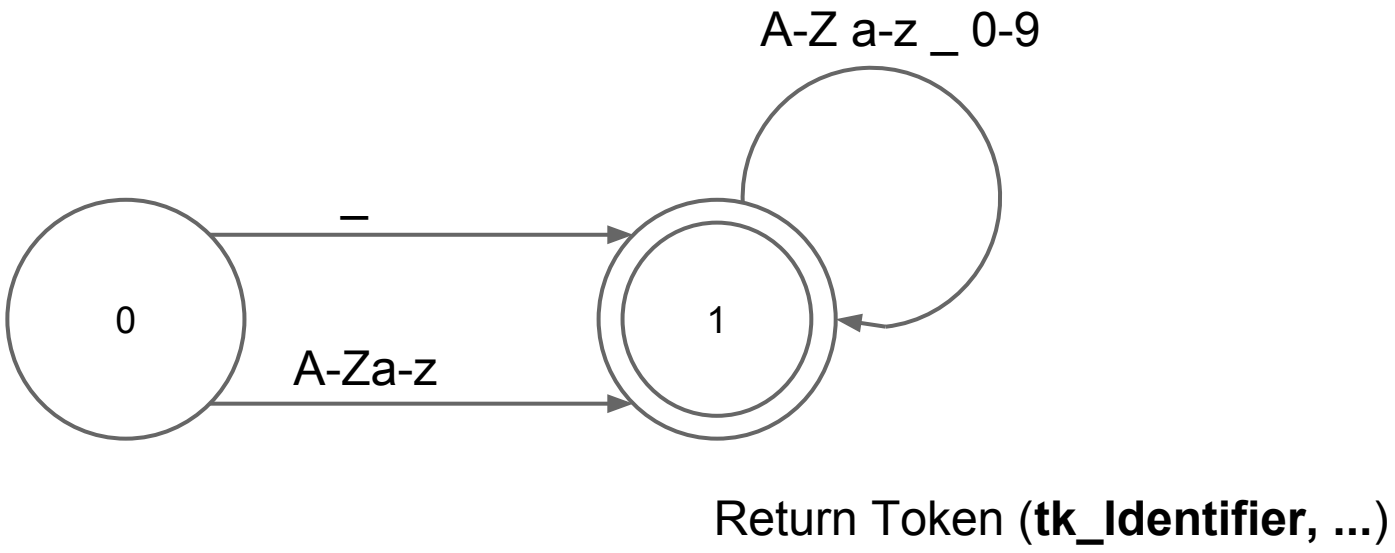
- Row
- Col
- Character Position

Lexical Analysis - Read from file

```
long Row, Col, Offset;

char Lexer::ReadChar() {
    char ch = read from file ( scanf / cin / ...)

    if( ch == '\n') { Row++; Col = 1; }
    else Col++;

    Offset++;
    return ch; }
```

Lexical Analysis - Categorisation

Identifier:



Return Token (**tk_Identifier, ...**)

Lexical Analysis - Categorisation

```
bool IsDigit ( char ch )

bool IsAlpha ( char ch )

bool IsIdentChar ( char ch )
```

Lexical Analysis - Categorisation

```
char ch = read next char
string strBuffer;

if( IsAlpha(ch) || ch == '_' )
    while( IsIdentChar(ch) ) {
        strBuffer += ch
        ch = read next char }

    // we read an identifier but ...
```

Extra Character

Ignoring it is not an option

Options:

1. Pushback
   a. Some APIs allow a character to be ‘unread’
   b. has to be read again

2. Store it
   a. next time round get it from the store
   b. use it as a look ahead

Look-Ahead

In many cases we cannot decide on what kind of token we have at hand from the current character only

We have to read more characters ahead

Examples:

Operators `==` `+=` `<=`

Backtracking

In some cases we have to backtrack and put back the lookahead character we have read

Example

Integer vs Real number

Token Lexer::**NextToken**()

```
ch = NextChar()
if ( IsWhitespace(ch) )
else if ( IsDigit(ch) ) … numbers
else if ( IsAlpha(ch) || ch == '_') …
else if ( IsOperatorChar() ) … operators
else if ( IsPunctuation(ch) ) ...
else
    return Token(tk_unknown, ...)
```

LexAn - Identifiers & Keywords

```
if( IsAlpha(ch) || ch == '_' ) {
    while( IsIdentChar(ch) ) {
        strBuffer += ch
        ch = read next char }
    Pushback(ch)
    if( lookup(strBuffer ) )
        token.Kind = tk_keyword
    else
        token.Kind = tk_identifier
    … }
```

LexAn - Numbers

```
if( IsDigit(ch) ) {
    while( IsDigit(ch) ) {
        strBuffer += ch
        ch = read next char }
    if( ch != '.' ) {
        Pushback(ch)
        return Token(tk_integer, …); }
    strBuffer += ch
    ...
```

LexAn - Numbers

```
ch = read next char
while( IsDigit(ch) ) {
    strBuffer += ch
    ch = read next char }
// Check for exponent
...
return Token(tk_real, …); }
```

Others

Strings
Characters
Comments

```
if( ch == '\"' ) {
    strBuffer += ch
    do
    {
        ch = read next char
        strBuffer += ch
    } while( ch != '\"' );
    return Token(tk_string, …); }
```

Motivation

Interesting

Interdisciplinarity

Lexical Analysis
- Compression

Parsing
- Data Loader
- Configuration

Optimisation

New Domains