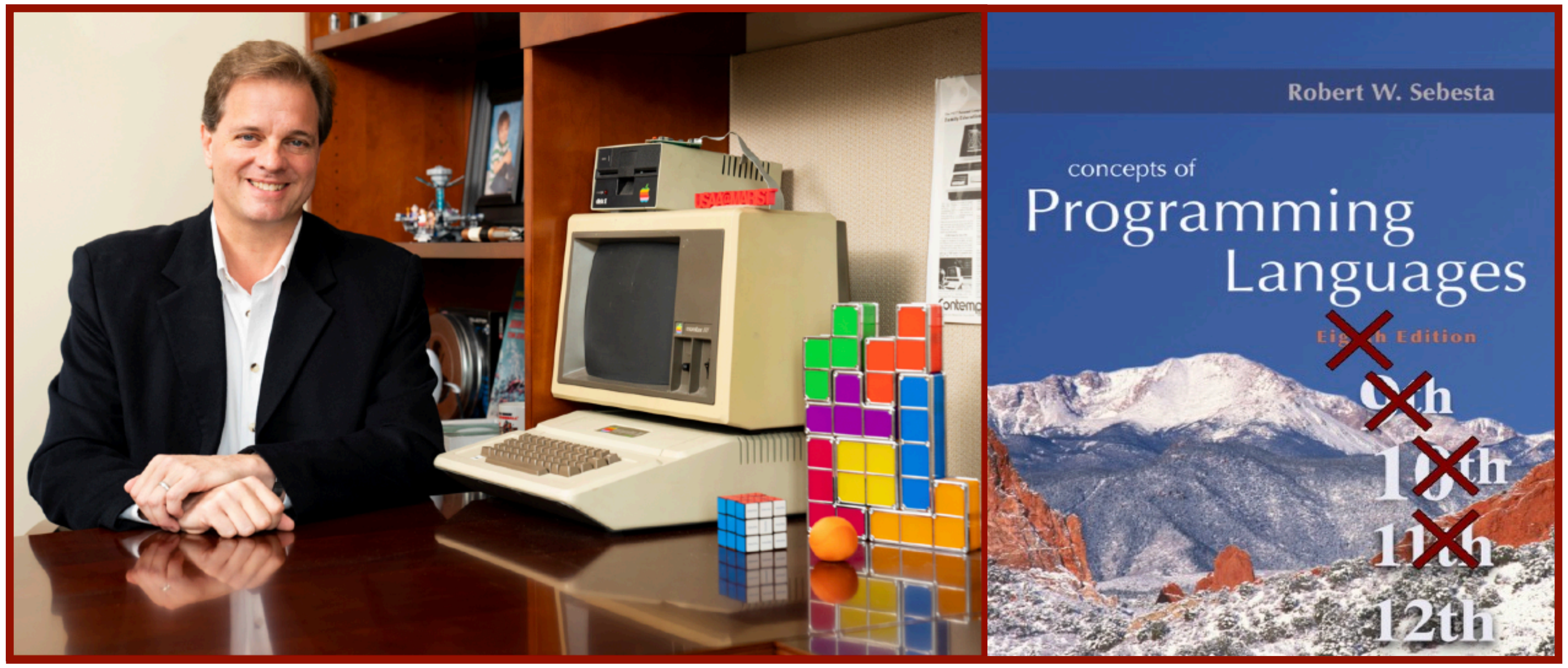
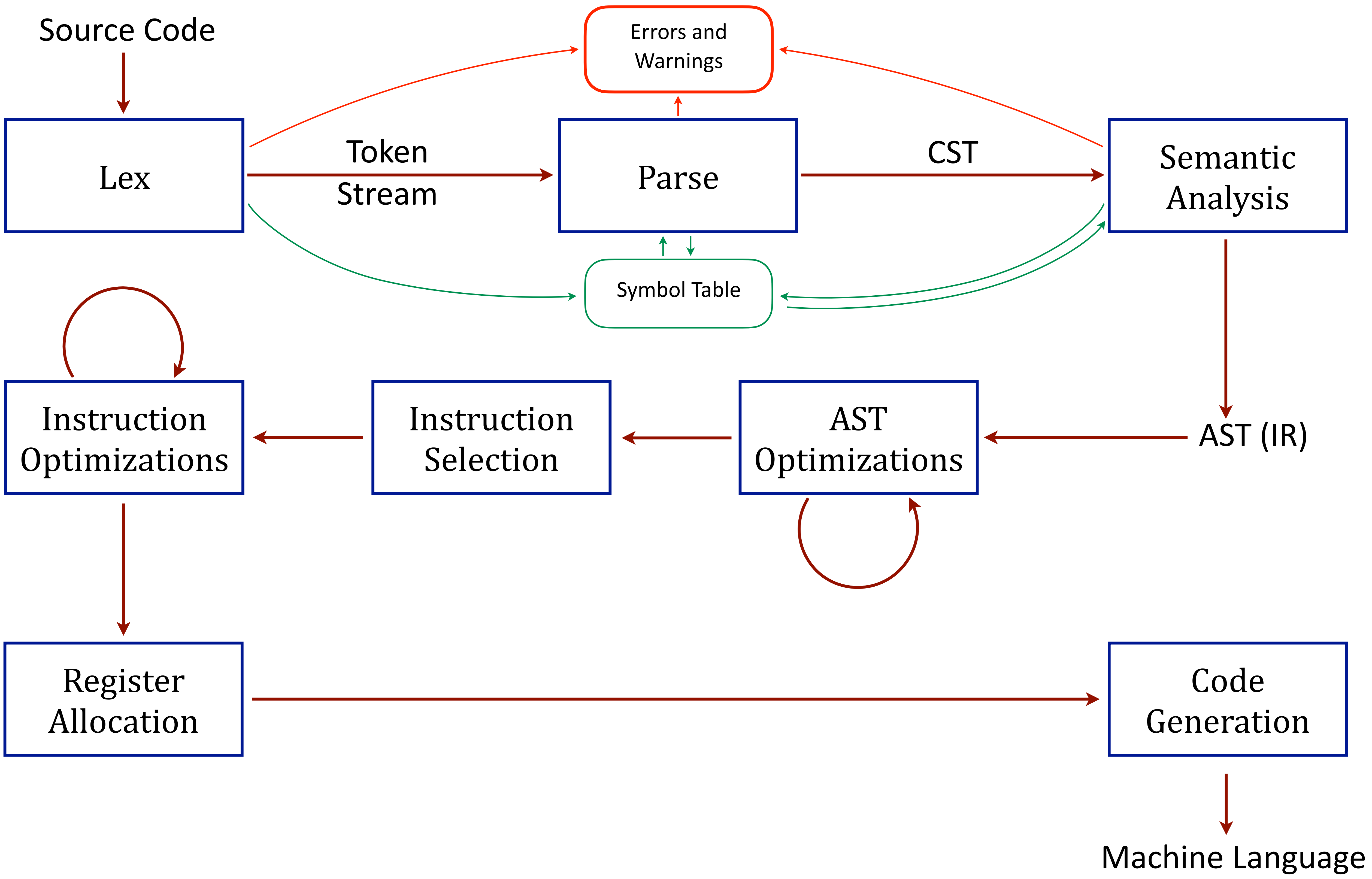
Lex and Parse

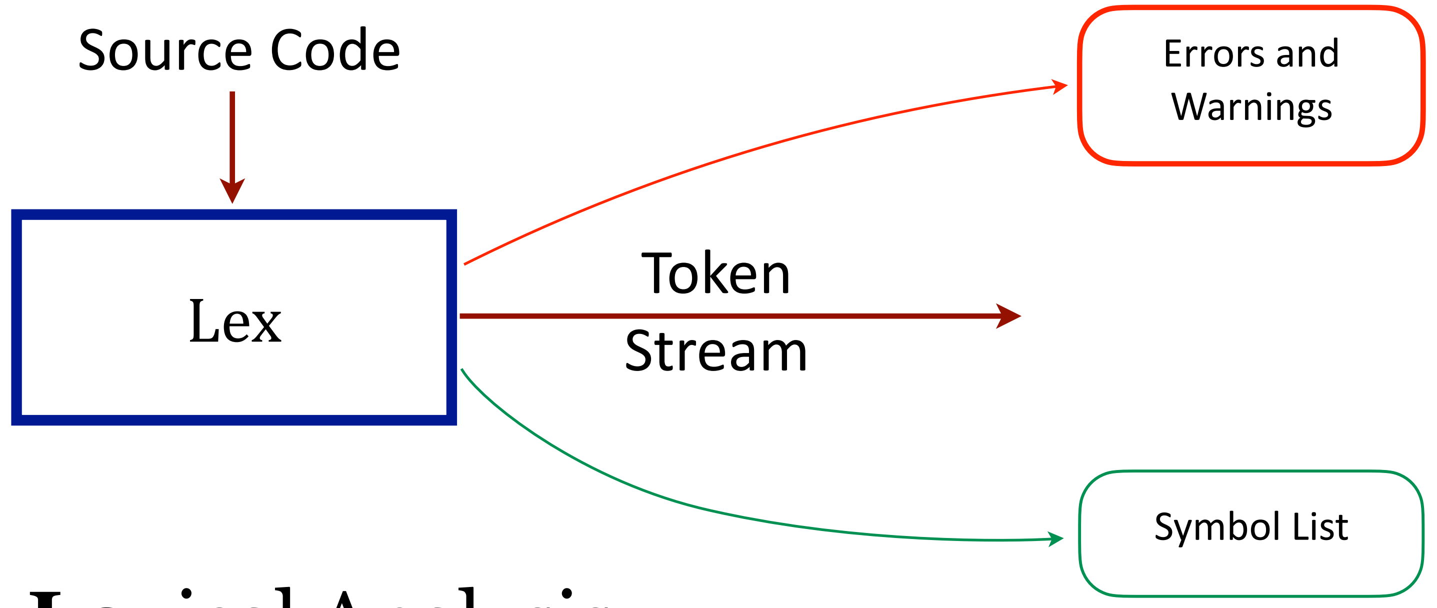


Alan G. Labouseur, Ph.D.
Alan.Labouseur@Marist.edu

Compiler Phases



Lexical Analysis



Lexical Analysis

- Maps characters into an ordered stream of tokens

  `x := x + y`

  becomes

  `<id,x> <assign> <id,x> <add> <id,y>`
- Typical tokens: `id int while print if`
- Eliminates white space and comments
- Reports meaningful errors and warnings
- Produces a token stream

- Focus on words/lexemes/tokens

Lexical Analysis

Only concerned with the words/syntax.

Not concerned with structure or meaning (sentences, type, scope).

Uses white space to help determine boundaries, then discards it.

Keeps track of line number for every token.

Builds the initial symbol list.

Definitions:

- A **token** is a sequence of characters that we'll treat as a unit in the grammar of our language.
- A **pattern** is a description of the form legal tokens can take.
- A **lexeme** is the sequence of characters in the source code that match a pattern for a token in the language.

Choosing Good Tokens

- Dependent on language.
- Varies by language.

- Typically . . .
  - keywords (different from identifiers)
  - identifiers (different from keywords)
  - punctuation symbols
  - digits
  - individual characters
- get their own tokens.

- Discard white space and comments, but only after making use of them if possible.

Choosing Good Tokens

Why is this important? What if we didn’t choose good tokens?

- PL/1 had no reserved words

  ```
  if then then then = else; else else = then;
  ```

  . . . so key words could be used as identifiers (variable and other names). Turns out that wasn’t great.

Choosing Good Tokens

Why is this important? What if we didn't choose good tokens?

- PL/1 had no reserved words

```
if then then then = else; else else = then;
```

- FORTRAN and Algol 68 ignored spaces, even for token identification

```
do 10 i = 1,25

do 10 i = 1.25
```

Choosing Good Tokens

Why is this important? What if we didn’t choose good tokens?

- PL/1 had no reserved words

```
if then then then = else; else else = then;
```

- FORTRAN and Algol 68 ignored spaces, even for token identification

```
do 10 i = 1,25
```
Loop from 1 to 25. (`10` is a label.)

```
do 10 i = 1.25
```
Assignment. (`do10i` is a variable.)

Regular Expressions

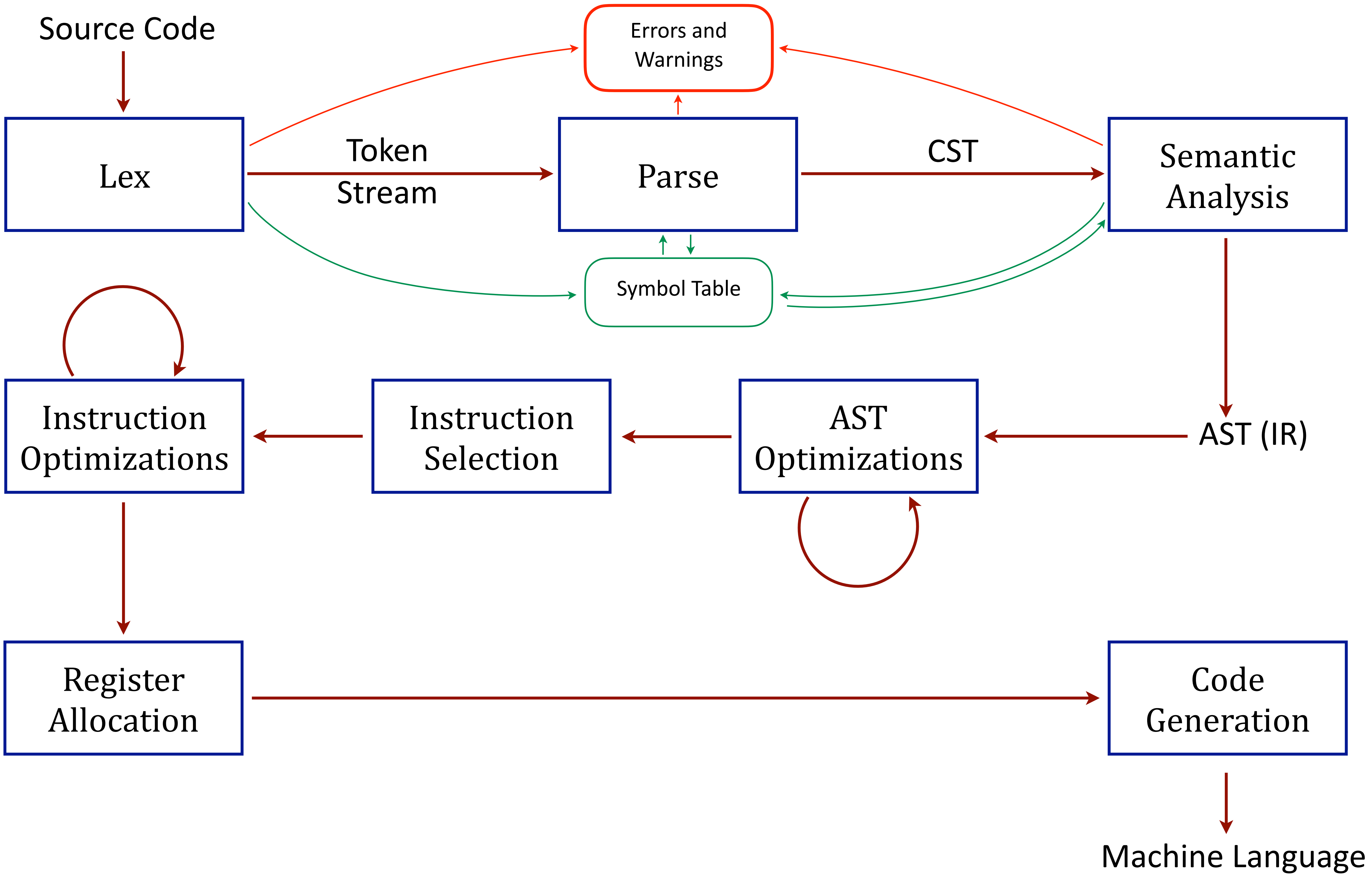
Let $\boldsymbol{\Sigma} = \{a, b\}$

• $a \mid b$ means "a or b".	$\{a, b\}$
• $(a \mid b)\ (a \mid b)$ means "a or b followed by a or b".	$\{aa, ab, ba, bb\}$
• a^* means "zero or more as".	$\{\boldsymbol{\varepsilon}, a, aa, aaa, aaaa, \ldots\}$
• a^+ means "one or more as".	$\{a, aa, aaa, aaaa, \ldots\}$

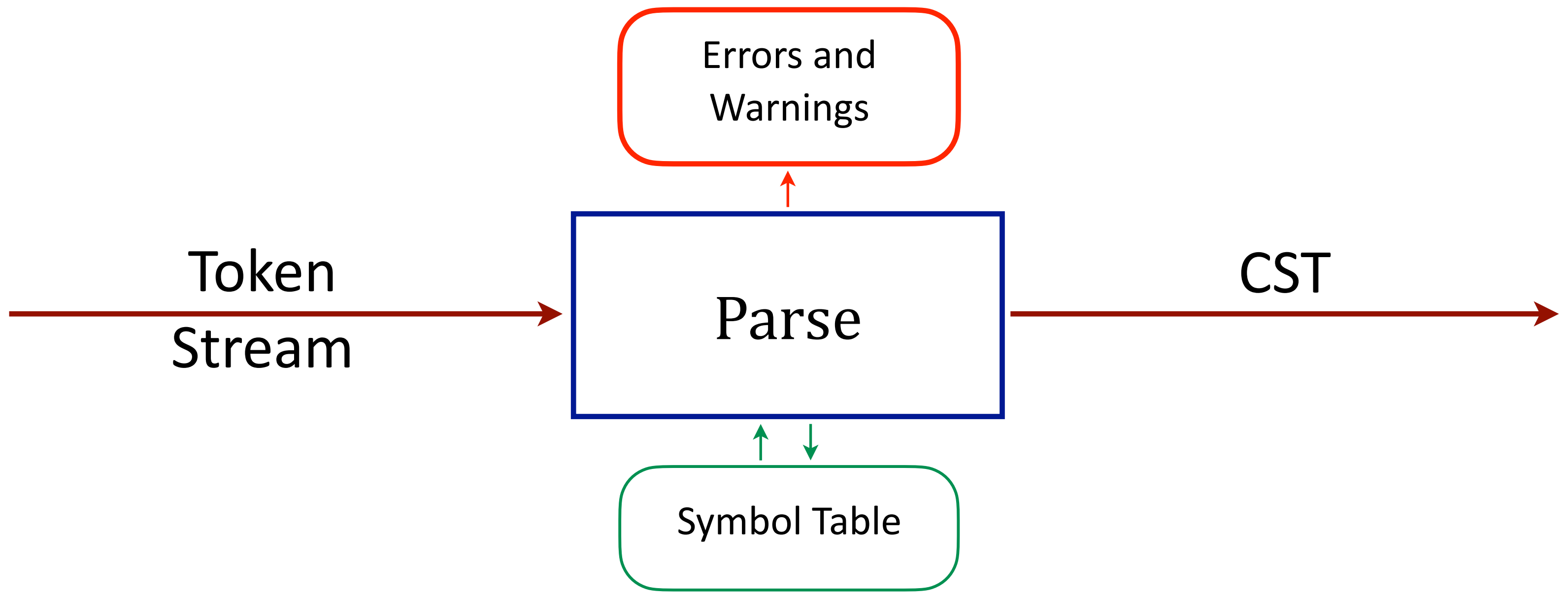
Other Examples

- $(a \mid b)^*$ means "all strings of as and bs including $\boldsymbol{\varepsilon}$".
- $(a^* \mid b^*)^*$ also means "all strings of as and bs including $\boldsymbol{\varepsilon}$".
- $a \mid (a^*b)$ denotes $\{a, b, ab, aab, aaab, aaaab, \ldots\}$

Compiler Phases



High Level View: Parse



Parse

- Recognize context-free syntax
- Recognize variables and type
- Report meaningful errors and warnings
- Produce a parse tree (aka: Concrete Syntax Tree)

- Focus on syntax

Predictive Parsing - LL(1) Grammar

Grammars that permit predictive parsing by reading the tokens
  Left-to-right, while doing a
  Left-most derivation, using only
  1 token look-ahead
are classified as LL(1) grammars.

Here's an example:

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

Remember ε?
It's the empty string.
This is an ε-production.

Predictive Parsing - LL(1) Grammar

Here's an LL(1) grammar:

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

Predictive parsing is easy* for LL(1) grammars. We can use a technique called **Recursive Descent**.

*Well, it's easy if you love recursion. But we all do, right?

Recursive Descent Parsing an LL(1) Grammar

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

To implement a Recursive Descent parser we need write a routine for each non-terminal in the grammar and a *match()* routine to consume tokens from the input.

```
parseStart()
parseValue()
parseExpr()
parseValues()

match()
```

Recursive Descent Parsing an LL(1) Grammar

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

```
parseStart() {
   parseValue()
   match($)
}
```

Recursive Descent Parsing an LL(1) Grammar

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

```
parseValue() {
   if token is num
      match(num)
   else
      match( ( )
      parseExpr()
      match( ) )
   end if
}
```

```
parseStart() {
   parseValue()
   match($)
}
```

Recursive Descent Parsing an LL(1) Grammar

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

```
parseExpr() {
   if token is +
      match(+)
      parseValue()
      parseValue()
   else
      match( * )
      parseValues()
   end if
}
```

```
parseStart() {
   parseValue()
   match($)
}

parseValue() {
   if token is num
      match(num)
   else
      match( ( )
      parseExpr()
      match( ) )
   end if
}
```

Recursive Descent Parsing an LL(1) Grammar

```
1.  <start>  → <value> $
2.  <value>  → num
3.           → ( <expr> )
4.  <expr>   → + <value> <value>
5.           → * <values>
6.  <values> → <value> <values>
7.           → ε
```

```
parseValues() {
  if token in {num,(}
      parseValue()
      parseValues()
   else
      // nothing
      // it's a ε
      // production
   end if
}
```

```
parseStart() {
   parseValue()
   match($)
}
```

```
parseValue() {
   if token is num
      match(num)
   else
      match( ( )
      parseExpr()
      match( ) )
   end if
}
```

```
parseExpr() {
   if token is +
      match(+)
      parseValue()
      parseValue()
   else
      match( * )
      parseValues()
   end if
}
```

Recursive Descent Parsing an LL(1) Grammar

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

```
parseValues() {
  if token in {num,(}
     parseValue()
     parseValues()
  else
     // nothing
     // it's a ε
     // production
  end if
}
```

```
parseStart() {
   parseValue()
   match($)
}
```

```
parseValue() {
   if token is num
      match(num)
   else
      match( ( )
      parseExpr()
      match( ) )
   end if
}
```

```
parseExpr() {
   if token is +
      match(+)
      parseValue()
      parseValue()
   else
      match( * )
      parseValues()
   end if
}
```

The <values> production begins with <value>. The first tokens in a <value> production are **num** and **(**.

Recursive Descent Parsing an LL(1) Grammar

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

```
parseValues() {
  if token in {num,(}
     parseValue()
     parseValues()
  else
     // nothing
     // it's a ε
     // production
  end if
}
```

The ε production is where the recursion stops. Even though there is no code, this is a great place for comments explaining what's going on.

```
parseStart() {
   parseValue()
   match($)
}

parseValue() {
   if token is num
      match(num)
   else
      match( ( )
      parseExpr()
      match( ) )
   end if
}
```

```
parseExpr() {
   if token is +
      match(+)
      parseValue()
      parseValue()
   else
      match( * )
      parseValues()
   end if
}
```

Recursive Descent Parsing an LL(1) Grammar

```
1. <start>  → <value> $
2. <value>  → num
3.          → ( <expr> )
4. <expr>   → + <value> <value>
5.          → * <values>
6. <values> → <value> <values>
7.          → ε
```

```
match( expectedTokens ) {
   if currentToken in expectedTokens
      consume currentToken
      inc tokenPointer
   else
      error: expected expectedTokens
             but found currentToken.
   end if
}
```

```
parseStart() {
   parseValue()
   match($)
}
```

```
parseValue() {
   if token is num
      match(num)
   else
      match( ( )
      parseExpr()
      match( ) )
   end if
}
```

```
parseExpr() {
   if token is +
      match(+)
      parseValue()
      parseValue()
   else
      match( * )
      parseValues()
   end if
}
```

```
parseValues() {
   if token in {num,(}
      parseValue()
      parseValues()
   else
      // nothing
      // it's a ε
      // production
   end if
}
```