EEX6335 – Compiler Design
EEX6363 – Compiler Construction

Day School – 4

by

Gehan Anthonys

Bachelor of Science Honours in Engineering
Bachelor of Software Engineering Honours

Department of Electrical and Computer Engineering
Faculty of Engineering
The Open University of Sri Lanka

13th September 2025

Lexical analysis (scanner) :

Main target → creation of a stream of tokens

by performing: removing white space, eliminating comments, recovering lexical errors (few number of), etc.

how design/organize → with

Lex specifications (valid tokens) → construct REs → NFA → DFA → State transition tables

Implemented (by table-driven, handwritten) -- lex/flex

TMA #1 and CAT #1

Example:

- Lexical specification –
  - Token list : <, <=, >, >=, ==, id, number, comment
  - Definitions:

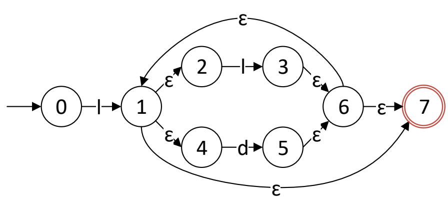
    id ::= letter(letter|digit)*
    number ::= digit*
    comment ::= % (letter|digit)* %

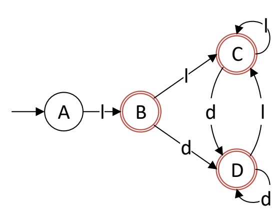
- Transform into a regular expression –

RE: (< | > | >= | <= | == | (l (l | d)*) | d* |%(l | d)*%))

E.g., *id* → **letter**(**letter**|**digit**)*

For the readability, lets denote **letter** ≡ l and **digit** ≡ d





A lexer defines how the contents of a file is broken into tokens.

A lexer is implemented as finite automata.

state	letter	digit	final
A	B		N
B	C	D	Y
C	C	D	Y
D	C	D	Y

Syntax analysis (parser) :

Main target → creation of Abstract syntax trees

by performing: Analyze the structure of the program & its component: *declarations, definitions, statements & expressions;* Check for, report, & recover from syntax errors, etc.

how design/organize → with

Tokens → Grammar → **Parse Tree** → SDT → **Abstract Syntax Tree**

In: Token stream/list; LL(1) grammar, first and follow sets
Out: Abstract syntax tree (via Syntax-Directed Translation)

Implemented -- top-down (LL) or bottom-up (LR):
(by yacc/bison, *ANTLR* or *JavaCC*)

TMA #2 and CAT #2

Parser -- all designs are based on a stack mechanism.

Top-down:
- predictive parsing, recursive descent, table-driven (requires removal of left recursions, ambiguities)

Bottom-up:
- simple LR (SLR),
- canonical LR (CLR),
- lookahead LR (LALR) (item generation)

A parser takes a token stream (emitted by a lexical analyzer) as input and based on the rules declared in the grammar (which define the syntax structure of the source) produces a parse tree data structure.

Table-driven predicting parsing method

```
E  → TE'
E' → ε | +TE'
T  → FT'
T' → ε | *FT'
F  → ( E ) | 0 | 1
```

```
FST(E)  : { 0,1,( }
FST(E') : { ε,+ }
FST(T)  : { 0,1,( }
FST(T') : { ε,* }
FST(F)  : { 0,1,( }
```

```
FLW(E)  : { $,) }
FLW(E') : { $,) }
FLW(T)  : { +,$,) }
FLW(T') : { +,$,) }
FLW(F)  : { *,+,$,) }
```

```
1. ∀p : ( (p ∊ R) ∧ (p : A→α) )
        do steps 2 and 3
2. ∀t : ( (t ∊ T) ∧ (t ∊ FIRST(α)) )
        add A→α to TT[A, t]
3. if ( ε ∊ FIRST(α) )
        ∀t : ( (t ∊ T) ∧ (t ∊ FOLLOW(A)) )
             add A→α to TT[A, t]
4. ∀e : ( (e ∊ TT) ∧ (e == ∅) )
        add "error" to e
```

```
r1: E  → TE'
r2: E' → +TE'
r3: E' → ε
r4: T  → FT'
r5: T' → *FT'
r6: T' → ε
r7: F  → 0
r8: F  → 1
r9: F  → (E)
```

LL(1) parsing table

	0	1	(	)	+	*	$
E	r1	r1	r1				
E'				r3	r2		r3
T	r4	r4	r4				
T'				r6	r6	r5	r6
F	r7	r8	r9				

How check the correctness?

E.g., Does the string 1*0 accept or not?

	0	1	(	)	+	*	$
E	r1	r1	r1				
E'				r3	r2		r3
T	r4	r4	r4				
T'				r6	r6	r5	r6
F	r7	r8	r9				

NOTE:
Empty cells are syntax errors.

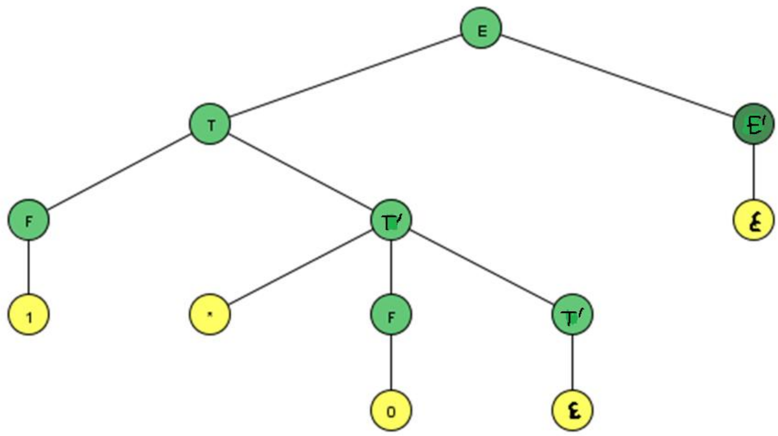
```
r1: E  → TE'
r2: E' → +TE'
r3: E' → ε
r4: T  → FT'
r5: T' → *FT'
r6: T' → ε
r7: F  → 0
r8: F  → 1
r9: F  → (E)
```

Derivation of 1*0

```
r1: E → TE'
r4:   → FT'E'
r8:   → 1T'E'
r5:   → 1*FT'E'
r7:   → 1*0T'E'
r6:   → 1*0E'
r3:   → 1*0
```

	Stack	Input	Production	Derivation
1	$E	1*0$		E
2	$E	1*0$	r1:E→TE'	⇒TE'
3	$E'T	1*0$	r4:T→FT'	⇒FT'E'
4	$E'T'F	1*0$	r8:F→1	⇒1T'E'
5	$E'T'1	1*0$		
6	$E'T'	*0$	r5:T'→ *FT'	⇒1*FT'E'
7	$E'T'F*	*0$		
8	$E'T'F	0$	r7:F→0	⇒1*0T'E'
9	$E'T'0	0$		
10	$E'T'	$	r6:T'→ε	⇒1*0E'
11	$E'	$	r3:E'→ε	⇒1*0
12	$	$		success

Now the parse tree for the string 1*0 can be drawn as



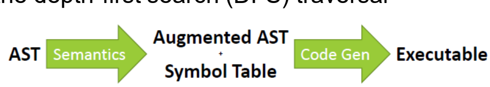
Semantic analysis:

Main target → creation of the symbol table

by performing: the storing of essential information about every symbol contained within the program.

how organize/design → with

- adds semantic information to the parse tree
- builds the symbol table.
- semantic actions and semantic records
- the depth-first search (DFS) traversal



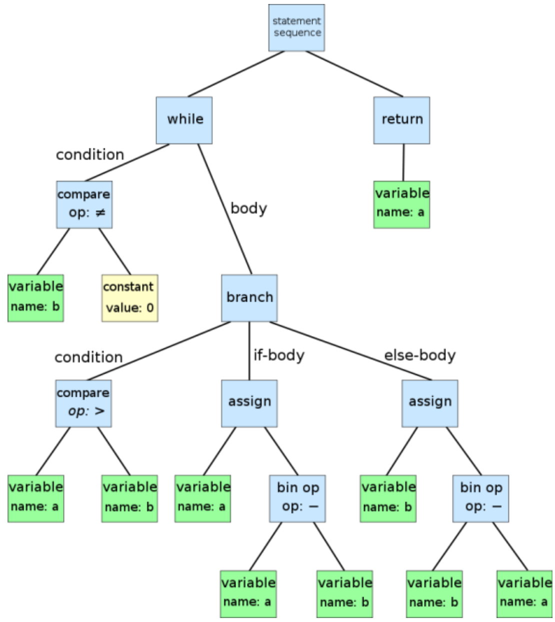
Implemented – via intermediate representations
(e.g., parse tree, abstract syntax tree…)
-- usually requires a complete parse tree,

TMA #3

Example 2:

An abstract syntax tree for the code below:

while b ≠ 0
    if a > b
        a := a − b
    else
        b := b − a
return a



Semantic rules and symbol table

Grammar Rule	Semantic Rules
Rule 1	Associated attribute equations
.	.
.	.
.	.
Rule n	Associated attribute equations

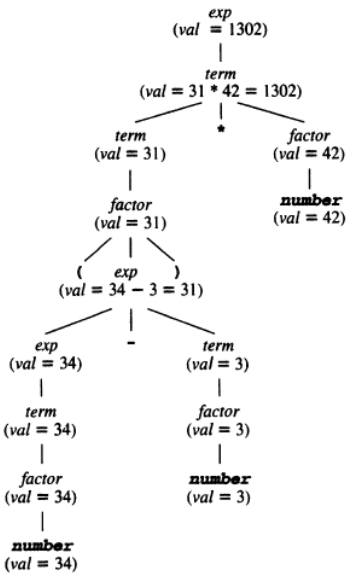
Reference:
- Louden, Kenneth C. "Compiler construction." *Cengage Learning* (1997).

E.g., in chapter 6

$exp \rightarrow exp + term \mid exp - term \mid term$
$term \rightarrow term * factor \mid factor$
$factor \rightarrow (\; exp \;) \mid \mathbf{number}$

Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2 + term$	$exp_1.val = exp_2.val + term.val$
$exp_1 \rightarrow exp_2 - term$	$exp_1.val = exp_2.val - term.val$
$exp \rightarrow term$	$exp.val = term.val$
$term_1 \rightarrow term_2 * factor$	$term_1.val = term_2.val * factor.val$
$term \rightarrow factor$	$term.val = factor.val$
$factor \rightarrow (\; exp \;)$	$factor.val = exp.val$
$factor \rightarrow \mathbf{number}$	$factor.val = \mathbf{number}.val$

- Parse tree for (34 - 3)*42



Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2$ + $term$	$exp_1.val = exp_2.val + term.val$
$exp_1 \rightarrow exp_2$ - $term$	$exp_1.val = exp_2.val - term.val$
$exp \rightarrow term$	$exp.val = term.val$
$term_1 \rightarrow term_2$ * $factor$	$term_1.val = term_2.val * factor.val$
$term \rightarrow factor$	$term.val = factor.val$
$factor \rightarrow$ (exp)	$factor.val = exp.val$
$factor \rightarrow$ **number**	$factor.val =$ **number**$.val$

The symbol table –

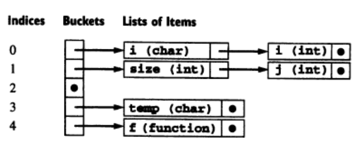
is used to store essential information about every symbol contained within the program. This includes:

- Keywords, • Data Types, • Operators,
- Functions, • Variable, * Procedures,
- Constants, • Literals , …

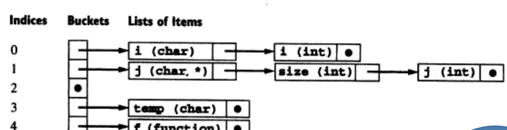
- E.g.,

- Reference: Louden, Kenneth C. "Compiler construction." *Cengage Learning* (1997).

```
int i,j;

int f(int size)
{ char i, temp;
  ...
  { double j;
    ...
  }
  ...
  { char * j;
    ...
  }
}
```

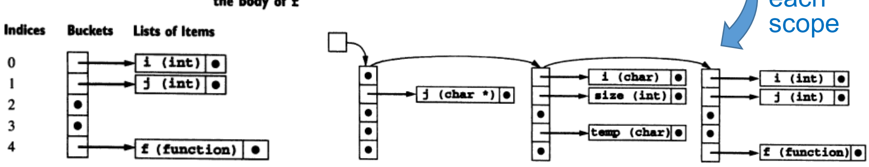


(a) After processing the declarations of the body of f



(b) After processing the declaration of the second nested compound statement within the body of f

Using separate tables for each scope



(c) After exiting the body of f (and deleting its declarations)

Symbol table: Global

name	kind	type	link
f1	function	float : int[2][2], float	
f2	function	int : nil	
MyClass1	class		
MyClass2	class		
program	function		

```
class MyClass1 {
  int mc1v1[2][4];
  float mc1v2;
  MyClass2 mc1v3[3];
  int mc1f1(int p1, MyClass2 p2[3]) {
    MyClass2 fv1[3];
    …
  }
  int f2(MyClass1 f2p1[3]) {
    int mc1v1;
    …
  }
}

class MyClass2 {
  int mc1v1[2][4];
  float fp1;
  MyClass2 m2[3];
  …
}

program {
  int m1;
  float[3][2] m2;
  MyClass2[2] m3;
  ...
}
float f1(int fp1[2][2], float fp2) {
  MyClass1[3] fv1;
  int fv2;
  …
}

int f2() {
...
}
```

Symbol table: f1

name	kind	type	link
fp1	parameter	int[2][2]	
fp2	parameter	float	
fv1	variable	MyClass1[3]	
fv2	variable	int	

Symbol table: f2

name	kind	type	link

Symbol table: MyClass1

name	kind	type	link
mc1v1	variable	int[2][4]	
mc1v2	variable	float	
mc1v3	variable	MyClass2[3]	
mc1f1	function	int : int, MyClass2[3]	
f2	function	int : MyClass1[3]	

Symbol table: MyClass1:mc1f1

name	kind	type	link
p1	parameter	int	
p2	parameter	MyClass2[3]	
fv1	variable	MyClass2[3]	

Symbol table: MyClass2

name	kind	type	link
mc1v1	variable	int[2][4]	
fp1	variable	float	
m2	variable	MyClass2[3]	

Symbol table: MyClass1:f2

name	kind	type	link
f2p1	parameter	MyClass1[3]	
mc1v1	variable	int	

Symbol table: program

name	kind	type	link
m1	variable	int	
m2	variable	float[3][2]	
m3	variable	MyClass2[2]	

Another representation of a symbol table

for the given code segment.

Code generation:

Main target → creation of the object/target code

by performing: the exact meaning of the source code and the efficient use of CPU and memory management.

how organize/design → with

- instruction selection and ordering, register allocation,
- identify the flow of values among the basic blocks by Directed Acyclic Graph (DAG) and represent by postfix
- 3AC (quadruples/ triples/ indirect triples)
- VM instructions (object/ stack-machine codes)

Implemented – via updating symbol table (adding an offset column)
-- using tag-based or stack-based methods
-- verifying the correctness with test cases.

DP

Reference: Louden, Kenneth C. "Compiler construction." *Cengage Learning* (1997).

Chapter 8:

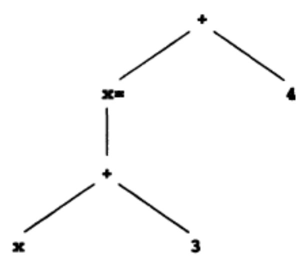
Example 1:

exp → **id** = *exp* | *aexp*
aexp → *aexp* + *factor* | *factor*
factor → (*exp*) | **num** | **id**

Grammar Rule	Semantic Rules
exp_1 → **id** = exp_2	exp_1 .name = exp_2 .name exp_1 .tacode = exp_2 .tacode ++ **id** .strval ‖ "=" ‖ exp_2 .name
exp → *aexp*	*exp* .name = *aexp* .name *exp* .tacode = *aexp* .tacode
$aexp_1$ → $aexp_2$ + *factor*	$aexp_1$ .name = newtemp() $aexp_1$ .tacode = $aexp_2$ .tacode ++ *factor* .tacode ++ $aexp_1$ .name ‖ "=" ‖ $aexp_2$ .name ‖ "+" ‖ *factor* .name
aexp → *factor*	*aexp* .name = *factor* .name *aexp* .tacode = *factor* .tacode
factor → (*exp*)	*factor* .name = *exp* .name *factor* .tacode = *exp* .tacode
factor → **num**	*factor* .name = **num** .strval *factor* .tacode = ""
factor → **id**	*factor* .name = **id** .strval *factor* .tacode = ""

The attribute of the target code for x = (x+3) + 4 is

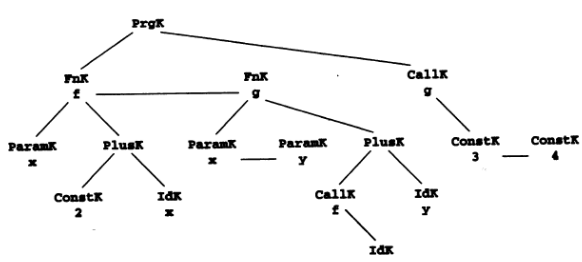
t1 = x +3
x = t1
t2 = t1 + 4



Example 2: Consider the grammar below on function definition and call

program → *decl-list exp*
decl-list → *decl-list decl* | ε
decl → **fn id** (*param-list*) = *exp*
param-list → *param-list* , **id** | **id**
exp → *exp* + *exp* | *call* | **num** | **id**
call → **id** (*arg-list*)
arg-list → *arg-list* , *exp* | *exp*

```
fn f(x)=2+x
fn g(x,y)=f(x)+y
g(3,4)
```



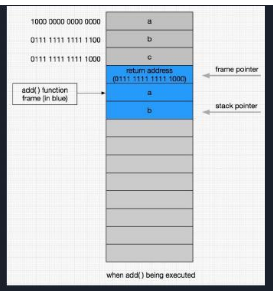

```
ent f
ldc 2
lod x
adi
ret
ent g
mst
lod x
cup f
lod y
adi
ret
mst
ldc 3
ldc 4
cup g
```

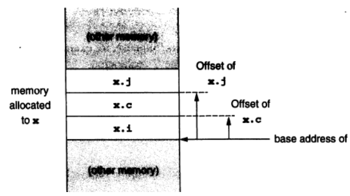
Offset example:

```
class MyClass {
    int x[3][8];
    int addNum() {
        int x;
    };
};

program {
    int x;
    int y;
    MyClass myClass[4][5];
    MyClass myClass1;

};
```

```
Table Name: MyClass table,  Parent Table Name: global table
--------------------------------------------------------------------------
| name         | kind         | type                  | offset | link          |
--------------------------------------------------------------------------
| addNum       | Function     | Int                   | 96     | addNum table  |
| x            | Variable     | Int[3][8]             | 0      | null          |
--------------------------------------------------------------------------
Table Name: program table,  Parent Table Name: global table
--------------------------------------------------------------------------
| name         | kind         | type                  | offset | link          |
--------------------------------------------------------------------------
| myClass1     | Variable     | MyClass               | 1928   | MyClass       |
| x            | Variable     | Int                   | 0      | null          |
| myClass      | Variable     | MyClass[4][5]         | 8      | null          |
| y            | Variable     | Int                   | 4      | null          |
--------------------------------------------------------------------------
```

```
int add(int a, int b) {
    return a + b;
}

program {
    int a;
    int b;
    int c;
    a = 1;
    b = 2;
    c = add(a, b);
    put c;
}
```





E.g.,
A symbol table with sizes/offsets

```
int func1(int int235[2][3][5]; float float4[10];){
  float float7;
  a1=a1+b1*1c1;
}
float func2(float float102[102]; float float6;){
  int int421[4][2][1];
  float float11[11];
  x2=a2+b2*c2;
  a3=x3+z3*y3;
}
program{
  float float101[101];
  float float4;
  int int4;
  float float3;
  int int3;
  a4=a4+b4*c4;
  x5=a5+b5*c5;
  a6=x6+z6*y6;
}
```

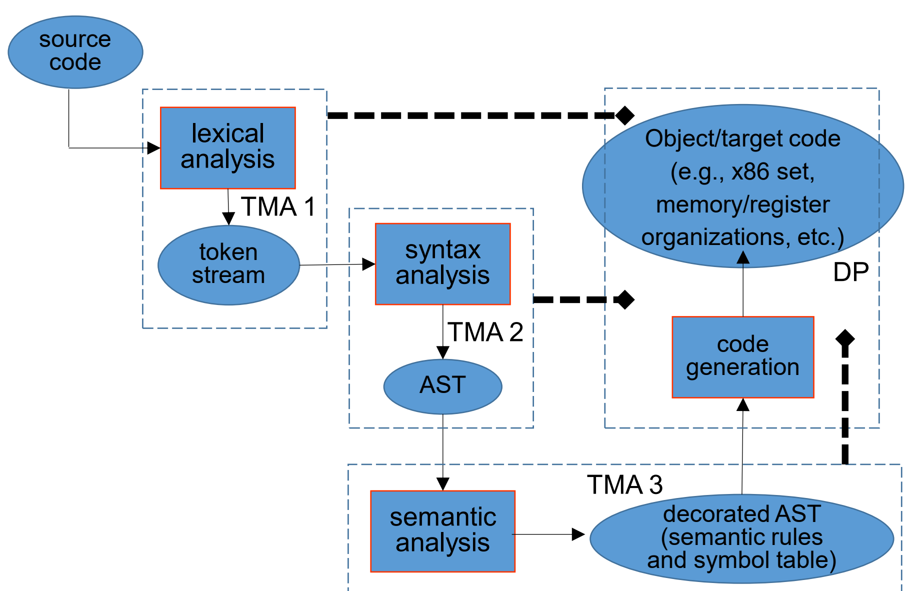
```
=====================================================
| table: global           scope offset: 0           |
=====================================================
| func       | func1      | int                     |
|    ==================================================
|    | table: func1            scope offset: -224    |
|    ==================================================
|    | var       | int235    | int        | 120 | -128 |
|    | var       | float4    | float      | 80  | -208 |
|    | var       | float7    | float      | 8   | -216 |
|    | tempvar   | t1        | int        | 4   | -220 |
|    | tempvar   | t2        | int        | 4   | -224 |
|    ==================================================
| func       | func2      | float                   |
|    ==================================================
|    | table: func2            scope offset: -964    |
|    ==================================================
|    | var       | float102  | float      | 816 | -828 |
|    | var       | float6    | float      | 8   | -836 |
|    | var       | int421    | int        | 32  | -868 |
|    | var       | float11   | float      | 88  | -956 |
|    | tempvar   | t3        | int        | 4   | -960 |
|    | tempvar   | t4        | int        | 4   | -964 |
|    | tempvar   | t5        | typeerror  | 0   | -964 |
|    | tempvar   | t6        | typeerror  | 0   | -964 |
|    ==================================================
| func       | program    | void                    |
|    ==================================================
|    | table: program          scope offset: -856    |
|    ==================================================
|    | var       | float101  | float      | 808 | -808 |
|    | var       | float4    | float      | 8   | -816 |
|    | var       | int4      | int        | 4   | -820 |
|    | var       | float3    | float      | 8   | -828 |
|    | var       | int3      | int        | 4   | -832 |
|    | tempvar   | t7        | int        | 4   | -836 |
|    | tempvar   | t8        | int        | 4   | -840 |
|    | tempvar   | t9        | int        | 4   | -844 |
|    | tempvar   | t10       | int        | 4   | -848 |
|    | tempvar   | t11       | int        | 4   | -852 |
|    | tempvar   | t12       | int        | 4   | -856 |
|    ==================================================
=====================================================
```

Example

```
program{
   int a;
   int b;
   int c;
   a = 1;
   put(a);
   b = 2;
   put(b);
   c = 3;
   put(c);
   a = a + b c;
   put(a + 6);
} // result = 13
```

```
==================================================================
| table: global              scope size: 0                       |
==================================================================
| func          | program    | void                              |
|     ==================================================================
|     | table: program              scope size: 40                     |
|     ==================================================================
|     | var          | a          | int        | 4     | 0             |
|     | var          | b          | int        | 4     | 4             |
|     | var          | c          | int        | 4     | 8             |
|     | litval       | t1         | int        | 4     | 12            |
|     | litval       | t2         | int        | 4     | 16            |
|     | litval       | t3         | int        | 4     | 20            |
|     | tempvar      | t4         | int        | 4     | 24            |
|     | tempvar      | t5         | int        | 4     | 28            |
|     | litval       | t6         | int        | 4     | 32            |
|     | tempvar      | t7         | int        | 4     | 36            |
|     ==================================================================
==================================================================
```


Goals of the assessments:



EEX6335 – Compiler Design
EEX6363 – Compiler Construction

Day School – 5

by

Gehan Anthonys

Bachelor of Science Honours in Engineering

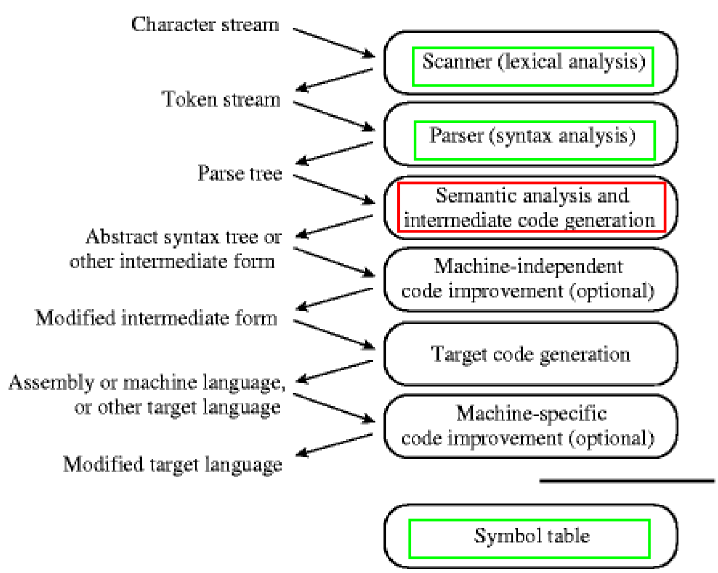
Bachelor of Software Engineering Honours

Department of Electrical and Computer Engineering

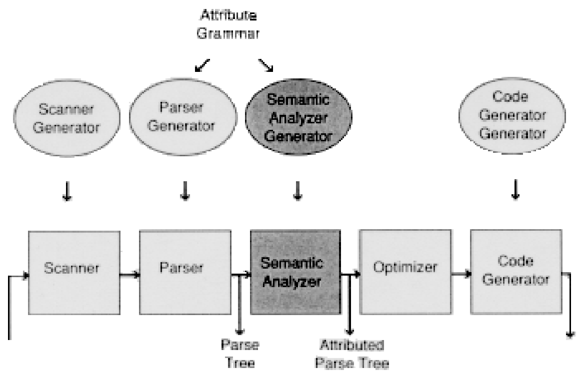
Faculty of Engineering

The Open University of Sri Lanka

Phases of Compilation



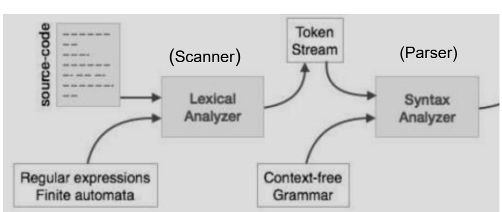
- Computer programs, like English sentences, have both a syntactic structure, illustrated by a parse tree, and a semantic structure which describes the meaning/actions.

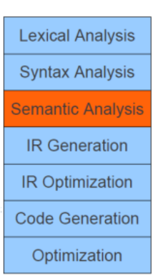


- Thus, the actions performed by the semantic analysis phase are a beginning of the process which will generate code.

• Two major actions: (1) it finishes the syntax analysis and also performs actions such as symbol table creation; (2) it translates the parse tree to an intermediate representation more appropriate for the later phases of optimization and code generation.

Let's consider detail analysis of semantic analyzer:





Semantic Analysis
- Categories
- Identifiers
- Symbol Table
- Rules

Semantic Analysis: the problem

Where we are?

- So far, we were able to check:
  - the program includes correct “words”;
  - “words” are combined in correct “sentences”;

Why Semantic Analysis?

- Lexically and syntactically correct programs may still contain other errors;
- Lexical and syntax analyses are not power enough to ensure the correct usage of variables, objects, functions, ...

What’s next?

We would like to:

- perform additional checks to increase guarantees of correctness;
- transform the program from the source language into the target one, and according to precisely defined semantic rules;

Additional Checks:

There are many additional checks that can be performed to increase correctness of code:

- Coherent (clear idea) usage of variables
  - definition-usage;
  - type;
- Existence of unreachable code blocks, . . .

Here, it is mainly focused on the mechanisms for **type checking** and generation of **intermediate code**

Semantic analysis:

- Ensure that the program satisfies a set of rules regarding the usage of programming constructs (variables, objects, expressions, statements & etc.,)
- The principal job of the semantic analyzer is to enforce static semantic rules.
- In general, anything that requires the compiler to compare things that are separate by a long distance or to count things ends up being a matter of *semantics.*
- The semantic analyzer also commonly constructs a syntax tree (usually first), and much of the information it gathers is needed by the code generator.

Syntax Directed Definitions:

- Attributes are used to associate characteristics and store values associated to grammar symbols.
- A syntax directed definition (SDD) provides the semantic rules to permit the definition of the values for the attributes.

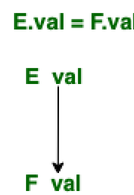
PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.code = E_1.code \| \| T.code \| \| ' + '$

- ➢ attributes are associated to grammar symbols and can be of any kind;
- ➢ rules are associated to productions.

An SDD can be defined using two different kinds of attributes:

- Synthesized attributes: a synthesized attributes at node N is defined only in terms of attribute values at the children of N and at N itself;
  - can be evaluated during a single bottom-up traversal of parse tree;
  - the production must have non-terminal as its head.
  - can be contained by both the terminals or non-terminals.

- Inherited attributes: an inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings;
  - can be evaluated during a single top-down and sideways traversal of parse tree.
  - the production must have non-terminal as a symbol in its body.
  - can't be contained by both, it is only contained by non-terminals.



Attribute Grammars

- Context-Free Grammars (CFGs) are used to specify the syntax of programming languages.

  E.g. arithmetic expressions

$$
\begin{aligned}
E &\longrightarrow E + T \\
E &\longrightarrow E - T \\
E &\longrightarrow T \\
T &\longrightarrow T * F \\
T &\longrightarrow T \,/\, F \\
T &\longrightarrow F \\
F &\longrightarrow - F \\
F &\longrightarrow (E) \\
F &\longrightarrow \texttt{const}
\end{aligned}
$$

How do we tie these rules to mathematical concepts?

- *Attribute grammars* are annotated CFGs in which *annotations* are used to establish meaning relationships among symbols
  - Annotations are also known as decorations

- Each grammar symbols has a set of *attributes*
  E.g. the value of E_1 is the attribute E_1.val

- Each grammar rule has a set of rules over the symbol attributes
  -- *Copy rules, Semantic Function rules*
  E.g. sum, quotient

Example

1: $E_1 \longrightarrow E_2$ + T
 ▷ E_1.val := sum (E_2.val, T.val)

2: $E_1 \longrightarrow E_2$ - T
 ▷ E_1.val := difference (E_2.val, T.val)

3: $E \longrightarrow T$
 ▷ E.val := T.val

4: $T_1 \longrightarrow T_2 * F$
 ▷ T_1.val := product (T_2.val, F.val)

5: $T_1 \longrightarrow T_2$ / F
 ▷ T_1.val := quotient (T_2.val, F.val)

6: $T \longrightarrow F$
 ▷ T.val := F.val

7: $F_1 \longrightarrow$ - F_2
 ▷ F_1.val := additive_inverse (F_2.val)

8: $F \longrightarrow$ (E)
 ▷ F.val := E.val

9: $F \longrightarrow$ const
 ▷ F.val := const.val

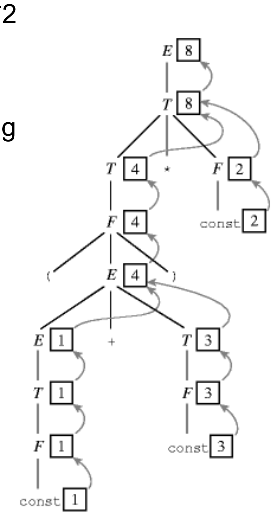
Attribute Flow

- The figure shows the result of annotating the parse tree for (1+3)*2

- Each symbols has at most one attribute shown in the corresponding box
  - Numerical value in this example
  - Operator symbols have no value

- Arrows represent *attribute flow*

Example



E	$\longrightarrow$	E + T
E	$\longrightarrow$	E - T
E	$\longrightarrow$	T
T	$\longrightarrow$	$T * F$
T	$\longrightarrow$	T / F
T	$\longrightarrow$	F
F	$\longrightarrow$	- F
F	$\longrightarrow$	(E)
F	$\longrightarrow$	const

Attribute Flow

Example

1: $E_1 \longrightarrow E_2 + T$
   ▷ E_1.val := sum (E_2.val, T.val)

2: $E_1 \longrightarrow E_2 - T$
   ▷ E_1.val := difference (E_2.val, T.val)

3: $E \longrightarrow T$
   ▷ E.val := T.val

4: $T_1 \longrightarrow T_2 * F$
   ▷ T_1.val := product (T_2.val, F.val)

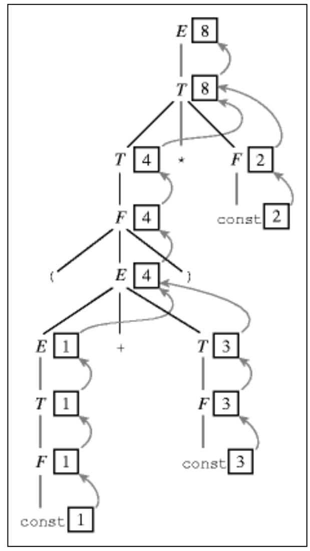
5: $T_1 \longrightarrow T_2 \ / \ F$
   ▷ T_1.val := quotient (T_2.val, F.val)

6: $T \longrightarrow F$
   ▷ T.val := F.val

7: $F_1 \longrightarrow - F_2$
   ▷ F_1.val := additive_inverse (F_2.val)

8: $F \longrightarrow (\ E \ )$
   ▷ F.val := E.val

9: $F \longrightarrow$ const
   ▷ F.val := const.val



Categories of Semantic Analysis

- Examples of semantic rules
  - Variables must be defined before being used
  - A variable should not be defined multiple times
  - In an assignment statement, the variable and the expression must have the same type
  - The test expression of an if statement must have Boolean type

- Two major categories
  - Semantic rules regarding types
  - Semantic rules regarding scopes

Type information and Type checking

- Type Information:
  Describes ***what kind of values*** correspond to different constructs: variables, statements, expressions, functions, etc.
  - variables: int a; integer
  - expressions: (a+1) == 2 boolean
  - statements: a = 1.0; floating-point
  - functions: int pow(int n, int m) int = int, int

- Type Checking:
  Set of rules which ensures the ***type consistency of different constructs*** in the program

Scope Information

- Characterizes the ***declaration of identifiers*** and the ***portions of the program*** where it is allowed to use each identifier
  E.g., identifiers: variables, functions, objects, labels

- Lexical scope: ***textual region*** in the program
  E.g.,
  Statement block, formal argument list, object body, function or method body, source file, whole program

- Scope of an identifier:
  The lexical scope its **declaration refers to**.

Variable Scope

- Scope of variables in statement blocks:

```
{ int a;
   ...
        {int b;
        ...
        }
   ....
}
```

scope of variable a

scope of variable b

- Scope of global variables: current file
- Scope of external variables: whole program

Function Parameter and Label Scope

- Scope of formal arguments of functions:

```
int foo(int n) {
    ...
}
```

scope of argument n

- Scope of labels:

```
void foo() {
   ... goto lab;
   ...
   lab: i++;
   ... goto lab;
   ...
}
```

scope of label lab,

Note: in C, all labels have function scope regardless of where they are

Scope in Class declaration

- Scope of object fields and methods:

```
class A {
   public:
         void f() {x=1;}
         ...
   private:
         int x;
         ...
}
```

scope of variable x
and method f

Semantic Rules for Scopes

Main rules regarding scopes:

- Rule 1: Use each identifier only within its scope
- Rule 2: Do not declare identifier of the same kind with identical names more than once in the same lexical scope

```
class X {
   int X;
   void X(int X) {
         X: ...
         goto X;
   }
}
```

```
int X(int X) {
   int X;
   goto X;
   {
         int X;
         X: X = 1;
   }
}
```

Both are legal but **NOT** recommended!

Symbol Tables

- Semantic ***checks refer to properties of identifiers*** in the program – their scope or type
- Need an ***environment to store*** the information about identifiers = symbol table
- Each entry in the symbol table ***contains***:
  - _Name_ of an identifier
  - Additional info about identifier: _kind_, _type_, _constant?_

NAME	KIND	TYPE	ATTRIBUTES
foo	func	int,int → int	extern
m	arg	int	
n	arg	int	const
tmp	var	char	const

Scope Information

- How to ***capture the scope information*** in the symbol table?
- Idea:
  - There is a hierarchy of scopes in the program
  - Use similar hierarchy of symbol tables
  - One symbol table for each scope
  - Each symbol table contains the symbols declared in that lexical scope

Example

```
int x;

void f(int m) {
    float x, y;
    ...
    {int i, j; ....; }
    {int x; l: ...; }
}

int g(int n) {
    char t;
    ... ;
}
```

Global symtab

x	var	int
f	func	int → void
g	func	int → int

func f symtab

m	arg	int
x	var	float
y	var	float

func g symtab

n	arg	int
t	var	char

i	var	int
j	var	int

x	var	int
l	label	

Problem

Associate each definition of x with its appropriate symbol table entry

```
int x;
void f(int m) {
    float x, y;
    ...
    {int i, j; x=1; }
    {int x; l: x=2; }
}

int g(int n) {
    char t;
    x=3;
}
```

Global symtab

x	var	int
f	func	int → void
g	func	int → int

m	arg	int
x	var	float
y	var	float

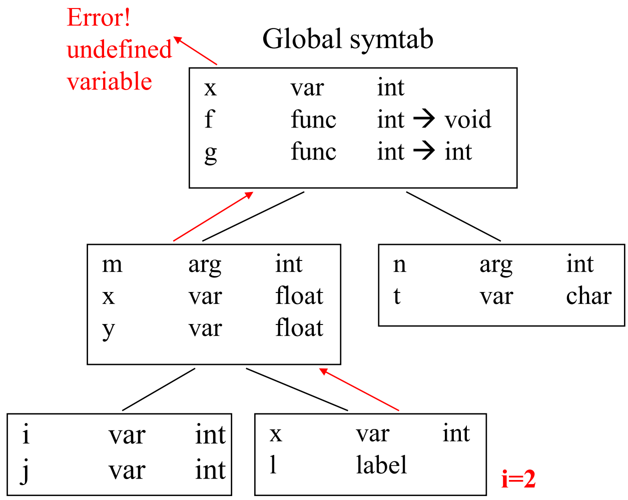
n	arg	int
t	var	char

i	var	int
j	var	int

x	var	int
l	label	

Catching Semantic Errors

```
int x;
void f(int m) {
   float x, y;
   ...
   {int i, j; x=1; }
   {int x; l: i=2; }
}

int g(int n) {
   char t;
   x=3;
}
```



Symbol Table Operations

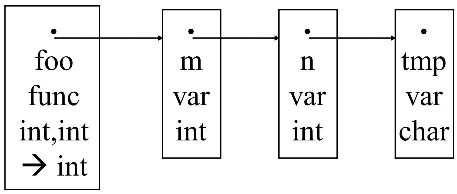
- Two operations:
  - To build symbol tables, we need to **insert new identifiers** in the table
  - In the subsequent stages of the compiler, we need to access the information from the table: **use lookup function**

- ***Cannot build symbol tables*** during lexical analysis
  - Hierarchy of scopes encoded in syntax

- ***Build the symbol tables***:
  - While **parsing**, using the semantic actions
  - After the AST is constructed

List Implementation

- Simple implementation using a list
  - One cell per entry in the table
  - Can grow dynamically during compilation



- Disadvantage: ***inefficient*** for large tables
  - Need to scan half the list on average

Hash table implementation

- Efficient implementation using hash table
  - Array of lists (buckets)
  - Use a hash on symbol name to map to corresponding bucket
    - Hash func: identifier name (string) → int
    - Note: include identifier type in match function

Forward References

- Use of an identifier within the scope of its declaration, but ***before it is declared***
- Any compiler phase that uses the information from the symbol table must be performed ***after*** the table is constructed
- Cannot type-check and build symbol table at the ***same time***

E.g.,

```
class A {
    int m() {return n(); }
    int n() {return 1; }
}
```

Type Checking

- Semantic checks to enforce the type safety of the program

- Examples
  - Unary and binary operators (e.g. +, ==, [ ]) must receive operands of the proper type
  - Functions must be invoked with the right number and type of arguments
  - Return statements must agree with the return type
  - In assignments, assigned value must be compatible with type of variable on LHS
  - Class members accessed appropriately

4 Concepts related to Types/Languages

1. Static vs dynamic checking -- When to check types
2. Static vs dynamic typing -- When to define types
3. Strong vs weak typing -- How many type errors
4. Sound type systems -- Statically catch all type errors

Static vs Dynamic Checking

- Static type checking
  - Perform at compile time
- Dynamic type checking
  - Perform at run time (as the program executes)
- Examples of dynamic checking
  - Array bounds checking
  - Null pointer dereferences

Static vs Dynamic Typing

Static and dynamic typing refer to type definitions

(i.e., bindings of types to variables, expressions, etc.)

- Static typed language
  - Types defined at compile-time and do not change during the execution of the program
    - C, C++, Java, Pascal
- Dynamically typed language
  - Types defined at run-time, as program executes
    - Lisp, Smalltalk

Strong vs Weak Typing

- Refer to how much type consistency is enforced
- Strongly typed languages -- Guarantee accepted programs are type-safe
- Weakly typed languages -- Allow programs which contain type errors
- These concepts refer to run-time -- Can achieve strong typing using either static or dynamic typing

Soundness

- Sound type systems: can statically ensure that the program is type-safe
- Soundness implies ***strong typing***
- Static type safety requires a conservative approximation of the values that may occur during all possible executions
  - May reject type-safe programs
  - Need to be expressive: reject as few type-safe programs as possible

Type Systems

- Type is predicate on a value
- Type expressions: Describe the possible types in the program
  - E.g., int, char*, array[], object, etc.
- Type system: Defines types for language constructs
  - E.g., expressions, statements

Type Expressions

- Language type systems have basic types (aka: primitive types or ground types) -- E.g., int, char*, double
- Build type expressions using basic types:
  - Type constructors
    - Array types; Structure types; Pointer types
  - Type aliases
  - Function types

Type Expressions: Arrays

- Various kinds of array types in different programming languages
- Array(T): arrays without bounds
  - C, Java: T[ ]
- Array(T,S): array with size
  - C: T[S], may be indexed 0 .. S-1
- Array(T,L,U): array with upper/lower bounds
  - Pascal: array[L .. U] of T
- Array(T, S1, …, Sn): multi-dimensional arrays
  - Fortran: T(L1, …, Ln)

Type Expressions: Structures / Functions

- Structures
  - Has form {id1: T1, … , idn: Tn} for some identifiers idi and types Ti
  - Is essentially cartesian product: (id1 x T1) x … x (idn x Tn)
  - Supports access operations on each field, with corresponding type
  - Objects: extension of structure types
- Functions
  - Type: T1 x T2 x … x Tn → Tr
  - Function value can be invoked with some argument expressions with types Ti, returns type Tr

Type Expressions: Aliases / Pointers

- Type aliases
  - C: typedef int int_array[ ];
  - Aliases are not type constructors
    - Int_array is the same type as int [ ]
  - Problem: Different type expressions may denote the same type
- Pointers
  - Pointer types characterize values that are addresses of variables of other types
  - C pointers: T* e.g., int *x;

Implementation

- Use a separate class hierarchy for types:
  - class BaseType extends Type {String name;}
  - class IntType extends BaseType { ... }
  - class FloatType extends BaseType { ... }
  - class ArrayType extends BaseType { ... }
  - class FunctionType extends BaseType { ... }

- Semantic analysis translates all type expressions to <u>type objects</u>

- Symbol table binds name to <u>type object</u>

Creating Type Objects

- Build types while parsing – use a syntax-directed definition

  - non terminal Type type
  - type : INTEGER
  - {$$ = new IntType(id); }
  - | ARRAY LBRACKET type RBRACKET
  - {$$ = new ArrayType($3); } ;

- Type objects are the abstract syntax tree (AST) nodes for type expressions

Type Checking

- Type checking is verify typing rules
  - E.g., “Operands of + must be integer expressions; the result is an integer expression”

Option 1: Implement using syntax-directed definitions (type-check during the parsing)

```
expr:   expr PLUS expr {
        if ($1 == IntType && $3 == IntType)
                $$ = IntType
        else
                TypeCheckError(“+”);
        }
```


Option 2: First build the AST, then implement type checking by recursive traversal of the AST nodes:

```
class Add extends Expr {
        Type typeCheck() {
                Type t1 = e1.typeCheck(), t2 = e2.typeCheck();
                if (t1 == Int && t2 == Int) return Int
                else TypeCheckError(“+”);
        }
}
```

Type Checking Identifiers

- Identifier expressions: Lookup the type in the symbol table

```
class IdExpr extends Expr {
        Identifier id;
        Type typeCheck() {return id.lookupType(); }
}
```

Type judgments for statements

- Statements may be expressions (i.e., represent values)
- Use type judgments for statements:
  - -- if (b) 2 else 3 : int
  - -- x == 10 : bool
  - -- b = true, y = 2 : int
- For statements which are not expressions: use a special unit type (void or empty type)
  - S : unit means "S is a well-typed statement with no result type"

•Deriving a Judgment

- Consider the judgment
  - if (b) 2 else 3 : int
- What do we need to decide that this is a well-typed expression of type int?
  - b must be a bool (b : bool); 2 must be an int (2 : int);
  - 3 must be an int (3 : int)

Static Semantics and Type Judgments

- Static semantics is the formal notation which describes type judgments:
  - E : T
  - means "E is a well-typed expression of type T"
- Type judgment examples:
  - -- 2 : int
  - -- true : bool
  - -- 2 * (3 + 4) : int
  - -- "Hello" : string
- Type judgment notation: $A \vdash E : T$
  - Means "In the context A, the expression E is a well-typed expression with type T"

Type context is a set of type bindings: id : T

(i.e. type context = symbol table)

$b: bool, x: int \vdash b: bool$

$b: bool, x: int \vdash if\ (b)\ 2\ else\ x : int$

$\vdash 2 + 2 : int$

Deriving a Judgment

- To show
  - $b: bool, x: int \vdash if\ (b)\ 2\ else\ x : int$
- Need to show
  - $b: bool, x: int \vdash b : bool$
  - $b: bool, x: int \vdash 2 : int$
  - $b: bool, x: int \vdash x : int$

Assignment Statements

$$\frac{id : T \in A \qquad A \vdash E : T}{A \vdash id = E : T}\ \text{(variable-assign)}$$

$$\frac{A \vdash E3 : T \qquad A \vdash E2 : int \qquad A \vdash E1 : array[T]}{A \vdash E1[E2] = E3 : T}\ \text{(array-assign)}$$

Sequence Statements

- Rule: A sequence of statements is well-typed if the first statement is well-typed, and the remaining are well-typed as well:

$$\frac{A \vdash S1 : T1 \qquad A \vdash (S2; .... ; Sn) : Tn}{A \vdash (S1; S2; .... ; Sn) : Tn}\ \text{(sequence)}$$

If Statements

- If statement as an expression: its value is the value of the clause that is executed

$$\frac{A \vdash E : bool \qquad A \vdash S1 : T \quad A \vdash S2 : T}{A \vdash \text{if } (E)\ S1 \text{ else } S2 : T} \quad \text{(if-then-else)}$$

- If with no else clause, no value, why??

$$\frac{A \vdash E : bool \qquad A \vdash S : T}{A \vdash \text{if } (E)\ S : unit} \quad \text{(if-then)}$$

Declarations

$$\frac{A \overset{\bullet}{\vdash} id : T\ [= E] : T1 \qquad A, id : T \overset{\bullet}{\vdash} (S2; \ldots ; Sn) : Tn}{A \overset{\bullet}{\vdash} (id : T\ [= E]; S2; \ldots ; Sn) : Tn} \quad \text{•(declaration)}$$

•= unit if no E

Declarations add entries to the environment (e.g., the symbol table)

Function Calls

- If expression E is a function value, it has a type
  T1 x T2 x ... x Tn → Tr
- Ti are argument types, where I = 1,2,…,n; and Tr is the return type
- How to type-check a function call? E(E1, ..., En)

$$\frac{A \vdash E : T1 \times T2 \times \ldots Tn \rightarrow Tr \qquad A \vdash Ei : Ti \quad (i \in 1 \ldots n)}{A \vdash E(E1, \ldots, En) : Tr} \quad \text{(function-call)}$$

Function Declarations

- Consider a function declaration of the form:
  - Tr fun (T1 a1, ... , Tn an) = E
  - Equivalent to:

    Tr fun (T1 a1, ..., Tn an) {return E;}
- Type of function body S must match declared return type of function, i.e., E : Tr
- But, in what type context?

Add arguments to environment

- Let A be the context surrounding the function declaration.
  - The function declaration: Tr fun (T1 a1, ... , Tn an) = E
  - Is well-formed if

    $$A, a1 : T1 , \ldots , an : Tn \ \vdash^{\bullet}\ E : Tr$$
- What about recursion?
  - Need: fun: T1 x T2 x ... x Tn → Tr ∈ A

Mutual Recursion

- Example
  - int f(int x) = g(x) + 1;
  - int g(int x) = f(x) – 1;
- Need environment containing at least
  - f: int → int, g: int → int
  - when checking both f and g
- Two-pass approach:
  - Scan top level of AST picking up all function signatures and creating an environment binding all global identifiers
  - Type-check each function individually using this global environment

How to check return?

- A return statement produces no value for its containing context to use
- Does not return control to containing context
- Suppose we use type unit ...
  - Then how to make sure the return type of the current function is T?

$$\frac{A \vdash E : T}{A \vdash \text{return } E : \text{unit}} \quad \text{(return)}$$

Put return in the Symbol table:

- Add a special entry {return_fun : T} when we start checking the function "fun", look up this entry when we hit a return statement
- To check Tr fun (T1 a1, ... , Tn an) { S } in environment A, need to check: $A, a1 : T1, ..., an : Tn, \text{return\_fun} : Tr \vdash A : Tr$

$$\frac{A \vdash E : T \qquad \text{return\_fun} : T \in A}{A \vdash \text{return } E : \text{unit}} \quad \text{(return)}$$

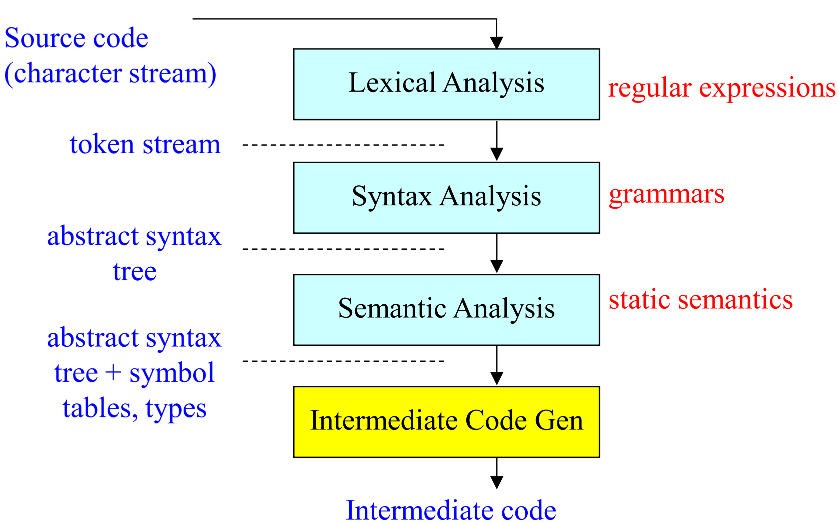
Static Semantics Summary

- Static semantics = formal specification of type-checking rules
- Concise form of static semantics: typing rules expressed as inference rules
- Expression and statements are well-formed (or well-typed) if a typing derivation (proof tree) can be constructed using the inference rules

Review of Semantic Analysis

- Check errors not detected by lexical or syntax analysis
- Scope errors
  - Variables not defined
  - Multiple declarations
- Type errors
  - Assignment of values of different types
  - Invocation of functions with different number of parameters or parameters of incorrect type
  - Incorrect use of return statements
- The most common way of *specifying* the semantics of a language is plain English
- There is a lack of formal rigor in the semantic specification of programming languages -- guess why

Where we are...



The code generator

- The code generator typically enters and uses detailed information about the storage assigned to identifiers.
- The following issues arise during the code generation phase :
    1. Input to code generator
    2. Target program
    3. Memory management
    4. Instruction selection
    5. Register allocation
    6. Evaluation order

- Each phase in a compiler can encounter errors.
- However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.

Error detection and Reporting:

- A compiler that stops when it finds the first error is not as helpful as it could be.
- The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler.
- Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase.
- The lexical phase can detect errors where the characters remaining in the input do not form any token of the language.
- After Syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program.
- We can think of this intermediate representation as a program for an abstract machine.

Intermediate form – Three-address code

- We consider an intermediate form called "three-address code (TAC or 3AC)," which every memory location can act like a register.
- The 3AC is an intermediate language that maps directly to "assembly pseudo-code", i.e. architecture-independent assembly code.
- It breaks the program into short uniform statements requiring no more than three variables (hence its name) and no more than one operator.
- As it is an intermediate (abstract) language, its "addresses" represent symbolic addresses (i.e. variables), as opposed to either registers or memory addresses that would be used by the target machine code.
- These characteristics allows 3AC to:
  - be more abstract than assembly language, enabling optimizations at the higher abstract level.
  - have high resemblance to assembly language, enabling easy translation to assembly language.

Three-address code

<table>
<tr><th>source</th><th>3AC</th></tr>
<tr><td><pre>x = a+b*c</pre></td><td><pre>   t := b*c
     x := a+t</pre></td></tr>
<tr><td><pre>x = (-b+sqrt(b^2-4*a*c))/(2*a)</pre></td><td><pre>   t1 := b * b
     t2 := 4 * a
     t3 := t2 * c
     t4 := t1 - t3
     t5 := sqrt(t4)
     t6 := 0 - b
     t7 := t5 + t6
     t8 := 2 * a
     t9 := t7 / t8
     x := t9</pre></td></tr>
<tr><td><pre>for (i = 0; i < 10; ++i) {
   b[i] = i*i;
}
...</pre></td><td><pre>   t0 := 0
L1: t1 = t1 >= 10
     if t1 goto L2
     t2 := t0 * t0
     t3 := t0 * 4
     t4 := b + t3
    *t4 := t2
     t0 := t0 + 1
     goto L1
L2: ...</pre></td></tr>
</table>

3AC	ASM

`t := b*c`	`L 3,b`
	`M 3,c`
	`ST 3,t`

`x := a+t`	`L 3,a`
	`A 3,t`
	`ST 3,x`

- The temporary variables are generated at compile time and may be added to the symbol table.
- In the generated code, the variables will refer to actual memory cells. Their address (or alias) may also be stored in the symbol table.

3AC	Quadruples
`t := b*c`	`MULT t,b,c`
`x := a+t`	`ADD x,a,t`

- 3AC can also be represented as **quadruples**, which are even more related to assembly languages.

Intermediate languages:

- In this case, we generate code in a language for which we already have a compiler or interpreter.
- Such languages are generally low-level and dedicated to the compiler construction task.
- It provides the compiler writer with a "virtual machine".
- Various compilers can be built using the same intermediate language and virtual machine.
- The virtual machine compiler can be compiled on different machines to provide a translator to various architectures.
- Many contemporary languages, such as *Java, Perl, PHP, Python* and *Ruby* use a similar execution architecture.

- For the project, you can use any processor's architecture, which provides a virtual assembly language and a compiler/interpreter for that language.

More examples:

```
entry   % ======program entry======
align   % following instruction align
addi    R1, R0, topaddr % initialize the stack pointer
addi    R2, R0, topaddr % initialize the frame pointer
subi    R1, R1, 12  % set the stack pointer to the top position of the stack
addi    R14, R0, 2  %
sw  -12(R2), R14    %
addi    R8, R0, 34  %
sw  -8(R2), R8  %
lw  R6, -12(R2) %
lw  R9, -8(R2)  %
lw  R11, -12(R2)    %
mul R9, R9, R11 %
add R6, R6, R9  %
sw  -4(R2), R6  %
lw  R10, -4(R2) %
putc    R10 %
hlt % ======end of program======
```

```
program {
    int x;
    int y;
    int z;
    x = 2;
    y = 34;
    z = x + y * x;
    put (z);
};
```



```
        entry   % ======program entry======
        align   % following instruction align
        addi    R1, R0, topaddr % initialize the stack pointer
        addi    R2, R0, topaddr % initialize the frame pointer
        subi    R1, R1, 4   % set the stack pointer to the top position of the stack
        addi    R14, R0, 65 %
        sw  -4(R2), R14 %
        lw  R8, -4(R2)  %
        ceqi    R8, R8, 1   %
        bz  R8, else_1  % if statement
        addi    R6, R0, 65  %
        sw  -4(R2), R6  %
        j   endif_1 % jump out of the else block
else_1  addi    R9, R0, 66  %
        sw  -4(R2), R9  %
endif_1 nop % end of the if statement
        lw  R11, -4(R2) %
        putc    R11 %
        hlt % ======end of program======
```

```
program {
    int x;
    x = 65;
    if (x == 1) then {
        x = 65;
    } else {
        x = 66;
    };
    put (x);
};
```

Variable declarations and value access/assignment:

- Integer variable declaration: `int x;`

```
x      res 4
```

where x is an alias to the fixed address of the memory cell containing the value of variable x. Such aliases are (unique) labels generated during the parse and stored in the symbol table.

Note that the labelling method of referring to values has great limitations, as it assumes that every variable in the program is unique and permanently and statically allocated.

- To load or change the content of an integer variable:

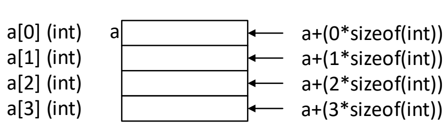
```
lw r1,x(r0)
sw x(r0),r1
```

where x is the label of variable x, r1 is the register containing the value of variable x and r0 is assumed to contain 0 (offset).

- Array of integers variable declaration:

```
int a[4];
```

```
a     res 16
```



  - *Element address = base address of the array + (offset number *number of bytes)*

- Accessing elements of an array of integers requires the use of offsets:

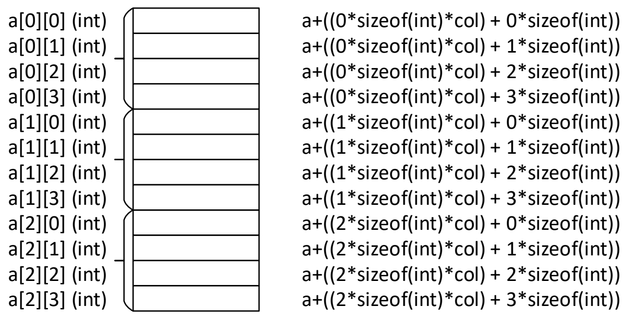
```
x = a[2];
```

```
addi r1,r0,8     % add immediate
lw r2,a(r1)      % load word
sw x(r0),r2      % store word
```

- Multidimensional arrays of integers:

```
int a[3][4];
```

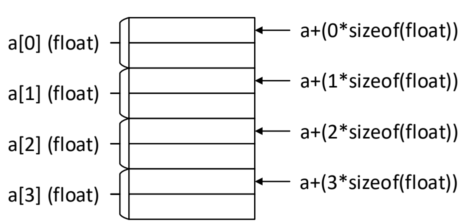
```
a      res 48
```



- To access specific elements, a more elaborated offset calculation needs to be implemented.

- For arrays of elements of aggregate type, or arrays where each element takes more than one memory cell:
  - The offset calculation needs to take into account to size of each element.

For example, assuming a float takes 8 bytes (2 words):

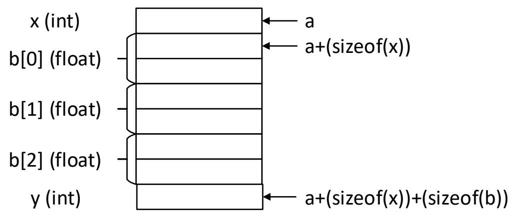



```
float f[4];
```

```
f     res 32
```

- For an object variable declaration, each data member is stored contiguously in the order in which it is declared.

```
class MyClass{
  int x;
  float b[3]
  int y;
}
```




```
Myclass a;
```

```
 a        res 32
```

- The offsets are calculated according to the total size of the data members preceding the member to access.

```
x = a.b[2]…
x = a + (sizeof(x)) + sizeof(float)*2)
```

```
addi r1,r0,4
addi r1,r1,16
lw r2,a(r1)
sw x(r0),r2
```


- Offsets can be pre-calculated in a previous phase and stored in the symbol table.
- This will eventually make code generation easier, and the generated code to be more concise.

```
=====================================================
| class     | MyClass                                |
|    =====================================================
|    | table: MyClass             scope size: 32        |
|    =====================================================
|    | var       | x      | int       | 4      | 0        |
|    | var       | b      | float     | 24     | 4        |
|    | var       | y      | int       | 4      | 28       |
|    =====================================================
|                                                      |
=====================================================
```

- Arithmetic operators

a+b

```
    lw r1,a(r0)
    lw r2,b(r0)
    add r3,r1,r2
t1  res 4
    sw t1(r0),r3
```

a+8

```
    lw r1,a(r0)
    addi r2,r1,8
t2  res 4
    sw t2(r0),r2
```

a*b

```
    lw r1,a(r0)
    lw r2,b(r0)
    mul r3,r1,r2
t3  res 4
    sw t3(r0),r3
```

a*8

```
    lw r1,a(r0)
    muli r2,r1,8
t4  res 4
    sw t4(r0),r2
```

- Relational operators

a==b

```
    lw r1,a(r0)
    lw r2,b(r0)
    ceq r3,r1,r2
t5  res 4
    sw t5(r0),r3
```

a==8

```
    lw r1,a(r0)
    ceqi r2,r1,8
t6  res 4
    sw t6(r0),r2
```


Code generation: suggested sequence

- Suggested sequence:
  - variable declarations (integers first)
  - expressions (one operator at a time)
  - assignment statement
  - read and write statements
  - conditional statement
  - loop statement

- Tricky parts:
  - function calls
  - expressions involving arrays and classes (offset calculation)
  - floating point numbers
  - function call stack
  - expressions involving access to object members (offset calculations)
  - calls to member functions (access to object’s data members)

Code generation:

Main target → creation of the object/target code

by performing: the exact meaning of the source code and the efficient use of CPU and memory management.

how organize/design → with

- instruction selection and ordering, register allocation,
- identify the flow of values among the basic blocks by Directed Acyclic Graph (DAG) and represent by postfix
- 3AC (quadruples/ triples/ indirect triples)
- VM instructions (object/ stack-machine codes)

Implemented – via updating symbol table (adding an offset column)
-- using tag-based or stack-based methods
-- verifying the correctness with test cases.

DP

Thank You

END.