Analysis Phases (front-end) (cont.)
مراحل التحليل – النهاية الأمامية

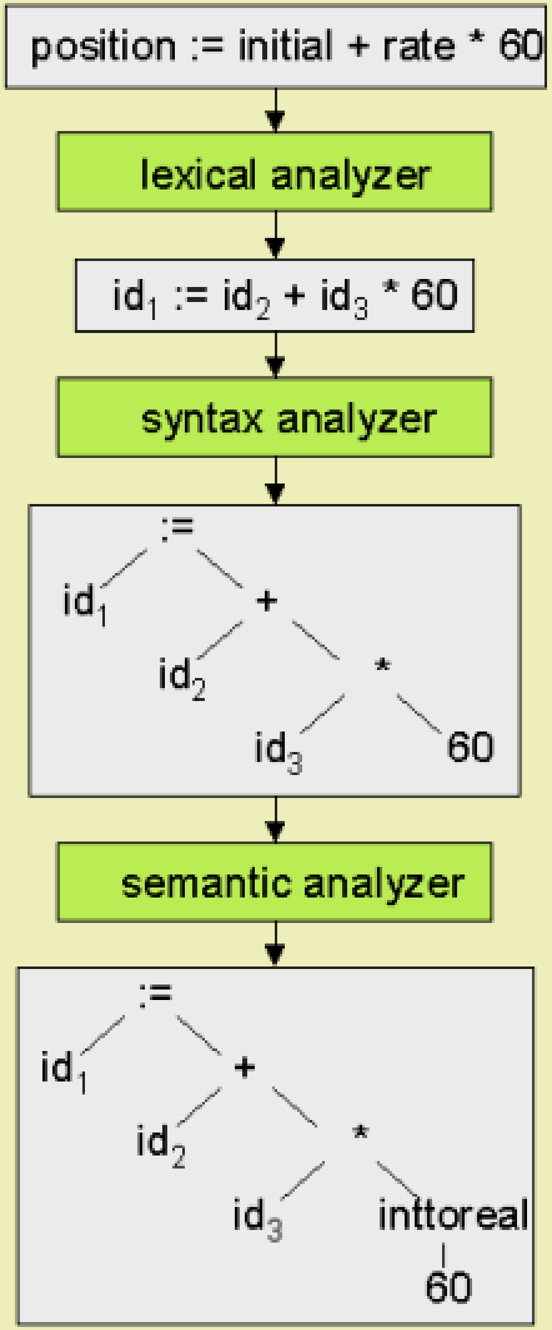
1-Scanner

2-Parser

3-Semantic analyzer

4-Intermediate code Generation

The Phases of a Compiler



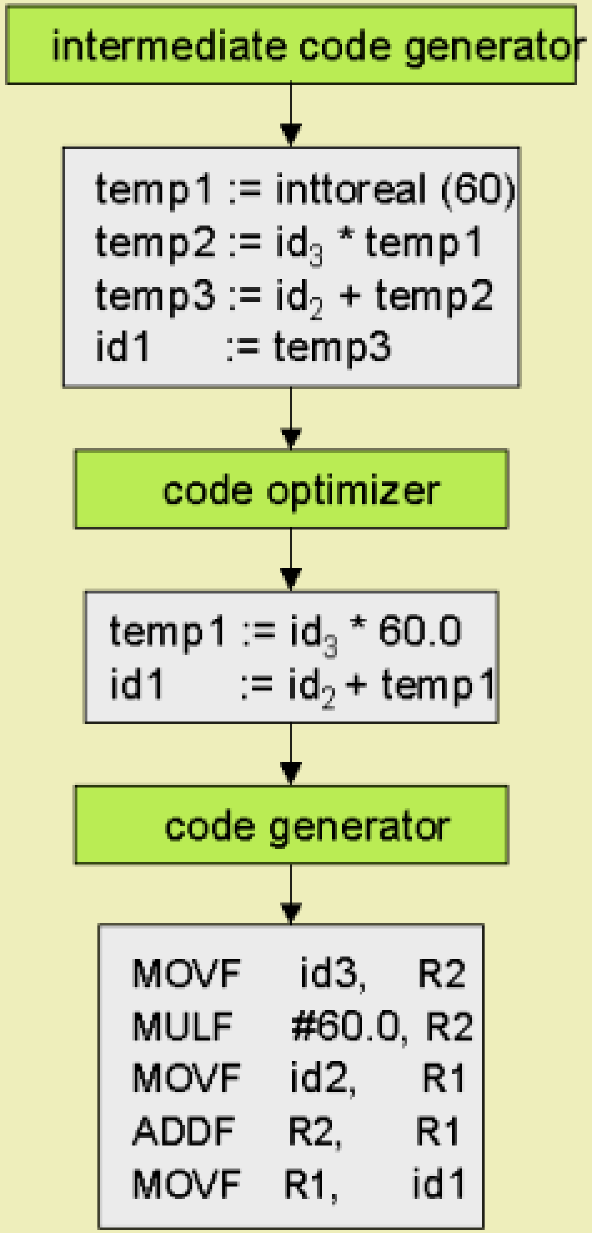
position := initial + rate * 60

lexical analyzer

$id_1 := id_2 + id_3 * 60$

syntax analyzer

semantic analyzer



intermediate code generator

temp1 := inttoreal (60)
temp2 := id_3 * temp1
temp3 := id_2 + temp2
id1 := temp3

code optimizer

temp1 := id_3 * 60.0
id1 := id_2 + temp1

code generator

```
MOVF    id3,    R2
MULF    #60.0,  R2
MOVF    id2,    R1
ADDF    R2,     R1
MOVF    R1,     id1
```

3-Semantic analyzer
المحلل اللغوي الدلالي

- هذا المحلل يهتم بالمعني مقارنة بالمحلل الإعرابي الذي يهتم بالنحو
- وعادة فإن التحليل الخاص بالمعني Semantic analyzer لأي برنامج يتم أثناء وقت التشغيل
- ولكن معظم لغات البرمجة لها خواص بحيث تستطيع تحديد هذه المعاني والدلالات قبل التشغيل

3-Semantic analyzer
المحلل اللغوي الدلالي

- وحيث أن هذه الخاصية لا تندرج تحت التحليل الإعرابي –فإن تلك الخاصية تسمي Static semantic (التحليل اللغوي الساكن) وهذه هي وظيفة Semantic analyzer
- ولكن التحليل اللغوي المتحرك Dynamic semantic هي خاصية للبرنامج والتي يتم تحديدها بواسطة تشغيل البرنامج
- وهذه لا يتم تحديدها بواسطة المترجم لماذا؟؟؟؟؟
- (لأن المترجم برنامج تحويل ينتهي عمله بالتحويل الي لغة الهدف فقط)

3-Semantic analyzer

- **Example of semantic analysis:**
- `int arr[2], c;`
- `c = arr * 10;`
- Most semantic analysis pertains to the checking of types. Although the C fragment above will scan into valid tokens and successfully match the rules for a valid expression, it isn't semantically valid.
  - معظم التحليل النوعي يعتمد علي تحقيق الأنواع ونلاحظ ذلك في التعبيرين السابقين

3-Semantic analyzer
المحلل اللغوي الدلالي

- In the semantic analysis phase, the compiler checks the types and reports that you cannot use an array variable in a multiplication expression and that the type of the right-hand-side of the assignment is not compatible with the left:

- ففي هذا التحليل فإن المترجم سيتحقق من الأنواع وسيقوم بتقديم تقرير متضمنا أنه لا يجوز أن يتضمن التعبير الرياضي اسم المصفوفة وبالتالي فإن طرفي المعادلة ليسوا متوافقين

4-Intermediate code Generation

- This is where the intermediate representation of the source program is created.
- وفي هذه المرحلة فإن التمثيل المتوسط لبرنامج المنبع سيتم عمله
- We want this representation to be easy to generate, and easy to translate into the target program.
- والمطلوب من هذا التمثيل البيني أن يكون سهلا في التكوين وسهلا في التحويل الي لغة الهدف

4-Intermediate code Generation

- The representation can have a variety of forms, but a common one is called *three-address code* (TAC) which is a lot like a generic assembly language,
- وهذا التمثيل ممكن أن يأخذ عدة اشكال ولكن الشكل المشترك والشائع هو ما يعرف بالكود ذو الثلاثة عناوين TACوالذي يشبه لغة الأسمبلي

4-Intermediate code Generation ***continue***

- Three-address code is a sequence of simple instructions, each of which can have at most three operands (one operator in addition to assignment). الكود ذو الثلاثة عناوين TAC عبارة عن تتابع من تعبيرات بسيطة وكل واحد يتكون من ثلاثة عوامل وفي وجود علامة رياضية وعلامة تخصيص
- The compiler must generate a temporary name to hold the value computed by each instruction ويقوم المترجم بتكوين اسم مؤقت ليحفظ القيمة المستنتجة من كل عملية

4-Intermediate code Generation

- **Example 1 of intermediate code generation:**
- `a = b * c + b * d    temp1 = b * c`
- `temp2 = b * d        temp3 = temp1 + temp2`
  `a =  temp3`
- The single C statement on the left is translated into a sequence of four instructions in three-address code on the right. والتعبير الموجود علي اليسار بلغة ال C تم تحويله الي تتابع من اربعة عمليات كل من ثلاثة اكواد في اليسار

4-Intermediate code Generation

- Note the use of temp variables that are created by the compiler as needed to keep the number of operands down to three.
  - لاحظ استخدام المتغيرات المؤقتة والتي تم توليدها بواسطة المترجم حيث يكون عدد العوامل أقل من او يساوي ثلاثة

4-Intermediate code Generation

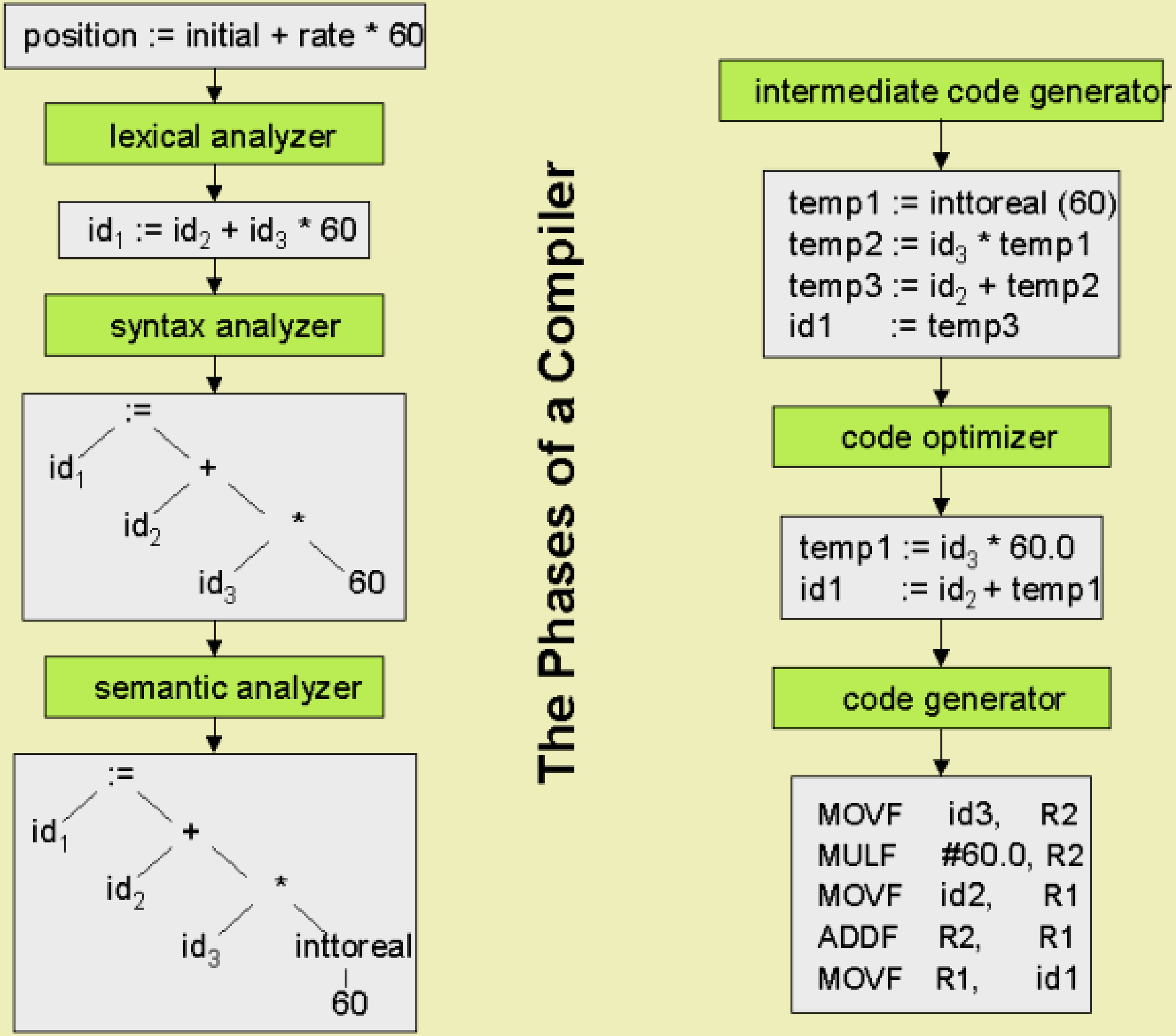
- Of course, it's a little more complicated than this, because we have to translate branching and looping instructions, as well as function calls.
- وطبعا الموضوع به بعض التعقيدات . فمثلا عند تحويل التفريعات والتكرارات وكذلك استدعاء الدوال والإجراءات

- Here is some TAC for a branching translation:
- والآتي مثال لتحويل تفريعة بواسطة if
- `if (a <= b)        _t1 = a > b`
- `a = a - c;        if _t1 goto L0`
- `c = b * c;        _t2 = a - c`
- `a = _t2        L0: _t3 = b * c`
  `c = _t3`

4-Intermediate code Generation

- Example 2 for the assignment statement :مثال ٢ في التعبير التخصيصي
- Position := initial + rate * 60 (1)
- Scanner o/p ➔ id1 := id2 + id3 * 60
- Syntax o/p ➔ fig. (1.5 a)
- Semantic o/p ➔ fig. (1.5 b)
- The source program in (1) might appear in TAC as :

The Phases of a Compiler



position := initial + rate * 60

lexical analyzer

$id_1 := id_2 + id_3 * 60$

syntax analyzer

semantic analyzer

intermediate code generator

temp1 := inttoreal (60)
temp2 := id_3 * temp1
temp3 := id_2 + temp2
id1 := temp3

code optimizer

temp1 := id_3 * 60.0
id1 := id_2 + temp1

code generator

```
MOVF    id3,    R2
MULF    #60.0,  R2
MOVF    id2,    R1
ADDF    R2,     R1
MOVF    R1,     id1
```

4-Intermediate code Generation

- وسيكون برنامج المصدر لهذا الجزء بالكود ذو الثلاثة عناوين TAC كالآتي
- temp1 := inttoreal(60)
- temp2 := id3 * inttoreal(60)
- temp3 := id2 + temp2
- id1 := temp3

The synthesis stages (back-end)
مراحل التركيب-النهاية الخلفية

*1) Intermediate Code Optimizatio*n

اختصار الأكواد المتوسطة

2) *Object Code Generatio*n

توليد الأكواد الموضعية

3) *Object Code Optimizatio*n:

اختصار الأكواد الموضعية

1) *Intermediate Code Optimization* اختصار الأكواد المتوسطة *(cont.)*

- The optimizer accepts input in the intermediate representation (e.g. TAC) and outputs a streamlined version still in the intermediate representation.
- وتقبل TAC كمدخلات وتسلسلات من التمثيل المتوسط كمخرجات تلك المرحلة التمثيل المتوسط
- In this phase, the compiler attempts to produce the smallest, fastest and most efficient running result by applying various techniques such as: وفي هذه المرحلة يقوم المترجم بمحاولة إنتاج أقل وأسرع أكواد بإستخدام تقنيات مثل

1) *Intermediate Code Optimization .cont* اختصار الأكواد المتوسطة

- • getting rid of unused variables.
- الاستفادة من المتغيرات الغير مستخدمة
- • eliminating multiplication by 1 and addition by 0
- حذف الضرب في واحد والجمع مع الصفر
- • loop optimization (e.g., remove statements that are not modified in the loop)
- إختصار التكرار وذلك بإزالة التعبيرات التي ليس لها دور في عملية التكرار
- • common sub-expression elimination.
- إزالة التعبيرات الصغيرة المشتركة

1) *Intermediate Code Optimization(cont.)* اختصار الأكواد المتوسطة

- The optimization phase can really slow down a compiler, so typically it is an optional phase.
- وحيث أن هذه المرحلة ربما تبطئ عمل المترجم فهي اختيارية
- The compiler may even have fine-grain controls that allow the developer to make tradeoffs between time spent compiling versus optimization quality
- وقد يكون في المترجم عملية تحكم لتقرير ما إذا كانت هذه المرحلة يمكن تنفيذها ام لا

Intermediate Code Optimization .cont
اختصار الأكواد المتوسطة

- **Example of code optimization:**
- `_t1 = b * c` `_t1 = b * c`
- `_t2 = _t1 + 0` `_t2 = _t1 + _t1`
- `_t3 = b * c` `a = _t2`
- `_t4 = _t2 +  _t3`
- `a =  _t4`

Intermediate Code Optimization .cont
اختصار الأكواد المتوسطة

- In the example shown above, the optimizer was able to eliminate an addition to the zero and a re-evaluation of the same expression, allowing the original five TAC statements to be re-written in just three statements and use two fewer temporary variables.
- وفي هذا المثال قامت الرحلة بإزالة الصفر وكذلك اختصار التعبيرات الي ثلاثة **فقط**

Intermediate Code Optimization .cont
اختصار الأكواد المتوسطة

- Example 2 for the assignment statement
- Position := initial + rate * 60 (1)
- Scanner o/p ➔ id1 := id2 + id3 * 60
- Syntax o/p ➔ fig. (1.5 a)
- Semantic o/p ➔ fig. (1.5 b)

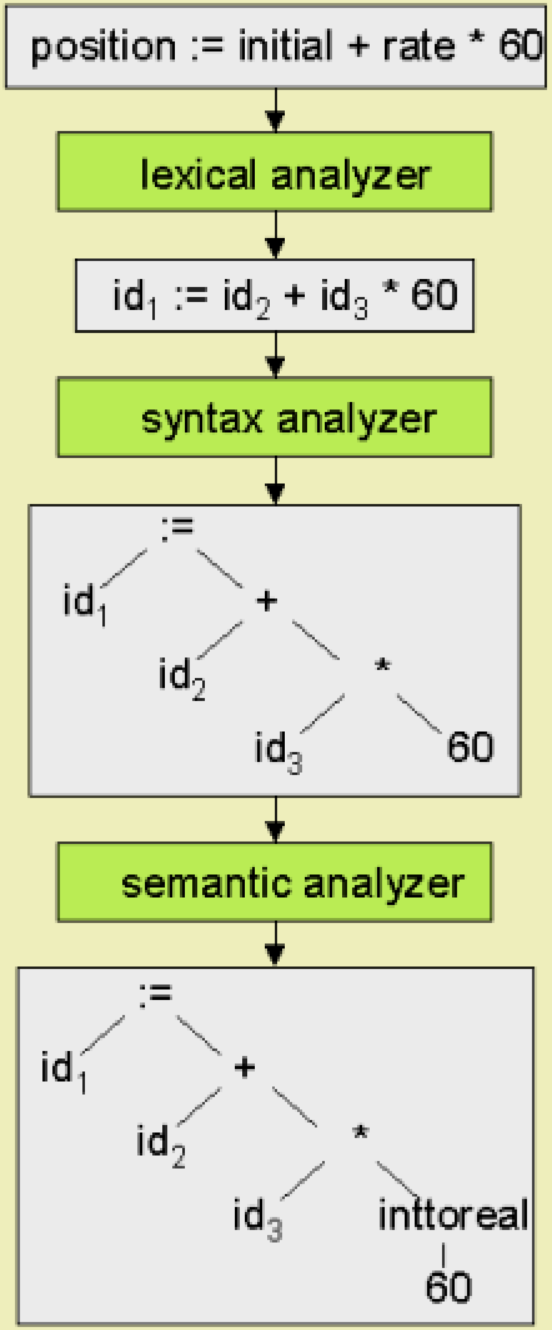
- Intermediate code generation ➔
- temp1 := inttoreal(60)
- temp2 := id3 * inttoreal(60)
- temp3 := id2 + temp2
- id1 := temp3

Code optimization >>

temp1 := id3 + 60.0

id1 := id2 + temp1 (2)

The Phases of a Compiler



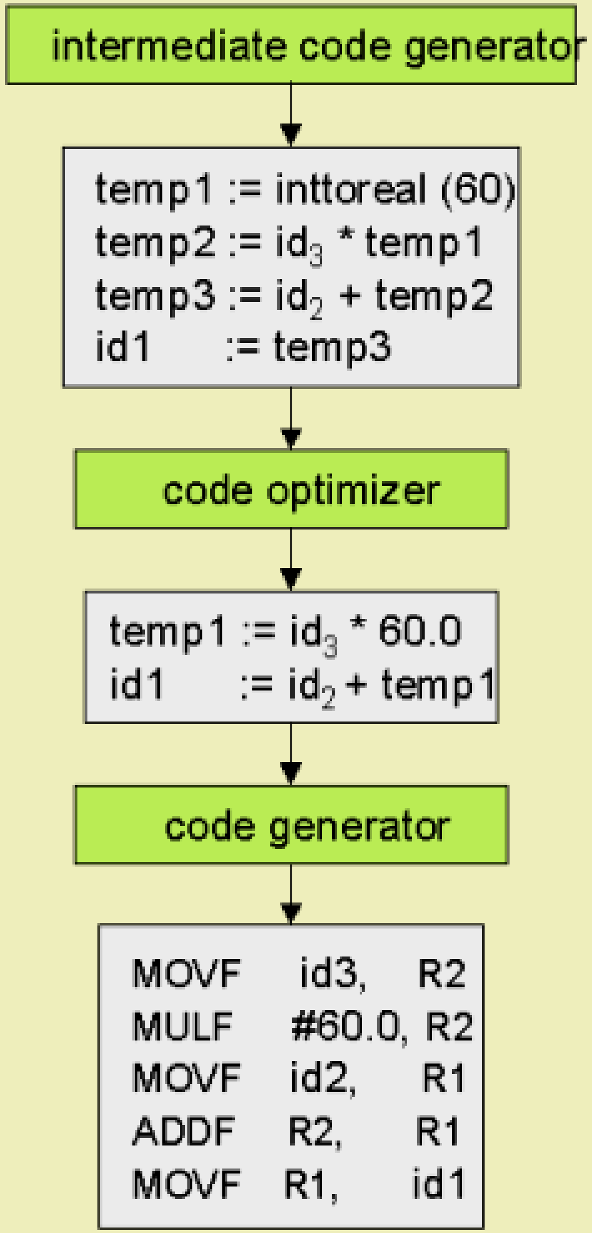
position := initial + rate * 60

lexical analyzer

$id_1 := id_2 + id_3 * 60$

syntax analyzer

semantic analyzer



intermediate code generator

temp1 := inttoreal (60)
temp2 := id_3 * temp1
temp3 := id_2 + temp2
id1 := temp3

code optimizer

temp1 := id_3 * 60.0
id1 := id_2 + temp1

code generator

```
MOVF    id3,    R2
MULF    #60.0,  R2
MOVF    id2,    R1
ADDF    R2,     R1
MOVF    R1,     id1
```

2) *Object Code Generatio*n
توليد الأكواد الموضعية

- This is where the target program is generated. The output of this phase is usually machine code or assembly code.
  - وفي هذه المرحلة يتم انتاج برنامج الهدف حيث يكون الخرج بلغة الأسمبلي أو لغة الآلة
- Memory locations are selected for each variable.
- يتم اختيار مكان لكل متغير في الذاكرة

2) *Object Code Generatio*n

توليد الأكواد الموضعية

- Instructions are chosen for each operation.
- وحيث يتم اختياراوامر لكل عملية
- The three-address code is translated into a sequence of assembly or machine language instructions that perform the same tasks.
  - ويتم تحويل الأكواد ذات الثلاثة TAC ألي تتابعات من أوامر لغة الأسمبلي أو لغة الآلة عناوين

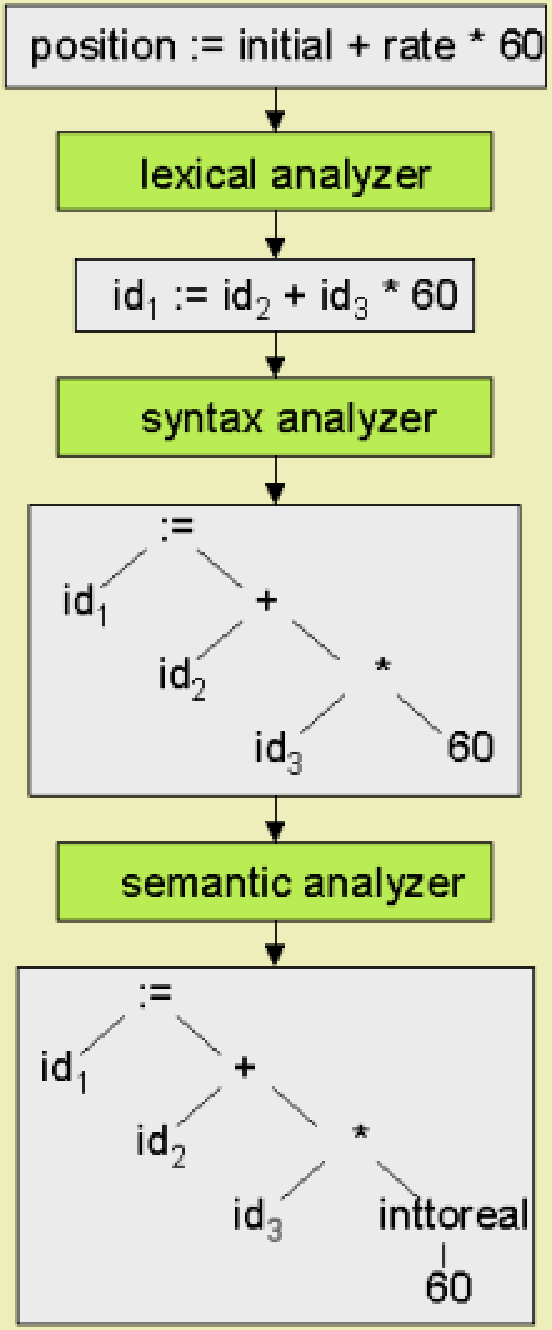
2) *Object Code Generatio*n
توليد الأكواد الموضعية

- **Example 2 where output of previous stage is:-**

```
temp1  :=  id3  +  60.0
 id1     :=  id2   +  temp1
```

- **for code generation and using registers R1 and R2 the translation of the code of (2) might become :**
- **MOVF id3,R2 MULF #60.0 ,R2**
- **MOVF id2 ,R1 ADDF R2 , R1**
- **MOVF R1 ,id1**

The Phases of a Compiler



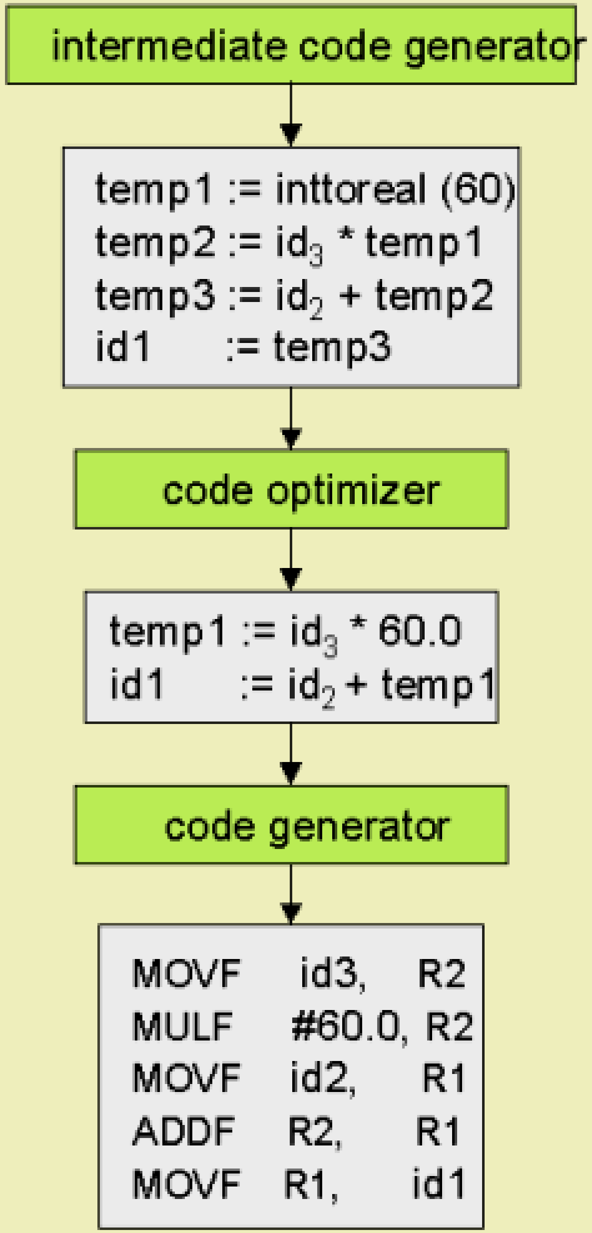
position := initial + rate * 60

lexical analyzer

$id_1 := id_2 + id_3 * 60$

syntax analyzer

semantic analyzer



intermediate code generator

temp1 := inttoreal (60)
temp2 := id_3 * temp1
temp3 := id_2 + temp2
id1 := temp3

code optimizer

temp1 := id_3 * 60.0
id1 := id_2 + temp1

code generator

```
MOVF    id3,   R2
MULF    #60.0, R2
MOVF    id2,   R1
ADDF    R2,    R1
MOVF    R1,    id1
```

3) *Object Code* optimization
إختصار الأكواد الموضعية

- There may also be another optimization pass that follows code generation, this time transforming the object code into tighter, more efficient object code.
  - ويوجد اختصار أخر بعد مرحلة توليد الأكواد حيث يتم تحويل الكود الموضوعي الي اخر اثر مرونة وكفاءة
-

3) *Object Code* optimization
إختصار الأكواد الموضعية

- This is where we consider features of the hardware itself to make efficient usage of the processor(s) and registers.
- للاستخدام الافضل ويتم في هذه المرحلة الأخذ في الإعتبار مكونات الحاسب الصلبة Hardware والاكفاء لل
- . processor(s) and registers.

3) *Object Code* optimization
إختصار الأكواد الموضعية

- The compiler can take advantage of machine-specific idioms (specialized instructions, pipelining, branch prediction, and other peephole optimizations) in reorganizing
- and streamlining the object code itself.
- This phase of the compiler (*Object Code Optimizatio*n) is usually configurable or can be skipped entirely.

The symbol table

- There are a few activities that interact with various phases across both stages.
- يوجد بعض النشاطات التي تتفاعل مع مراحل المترجم المختلفة
- One is *symbol table management*; a symbol table contains information about all the identifiers in the program along with important attributes such as type and scope.

The symbol table

- من هذه النشاطات ادارة جدول الرموز حيث يحتوي هذا الجدول علي معلومات عن جميع المعرفات في البرنامج وخواصها مثل النوع والمجال
- Identifiers can be found in the lexical analysis phase and added to the symbol table.
- هذه المتغيرات يتم التعامل معها في مرحلة المحلل الإملائي لذا يتم اضافتها للجدول في هذه المرحلة

The symbol table

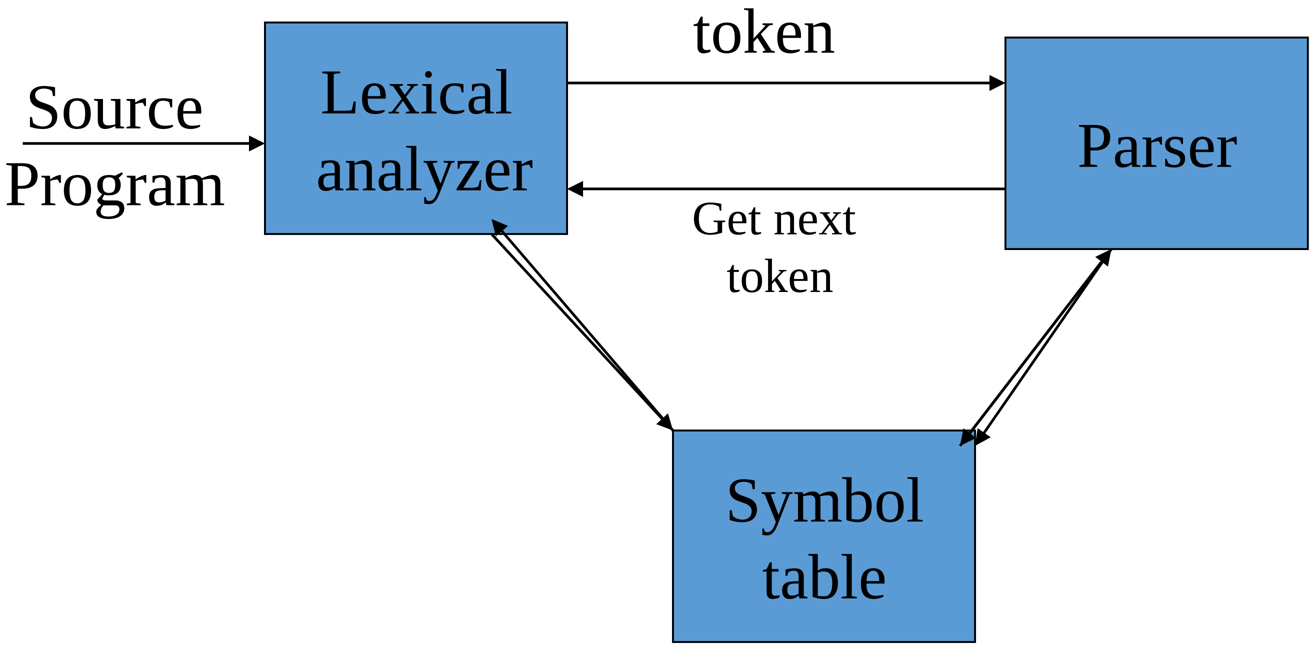
- During the two phases that follow (syntax and semantic analysis), the compiler updates the identifier entry in the table to include information about its type and scope.
- وفي المرحلتين syntax and semantic يقوم المترجم بتحديث التاليتين
- بيانات هذه المتغيرات
- When generating intermediate code, the type of the variable is used to determine which instructions to emit.

The symbol table

- During optimization, the “live range” of each variable may be placed in the table to aid in register allocation.
- وخلال مرحلة الاختصار فان المدي الفعلي لكل متغيرقد يتم وضعه في الجدول للمساعدة في تحديد المسجل المناسب
- The memory location determined in the code generation phase might also be kept in the symbol table.
- وتحديد مكان الذاكرة والذي يتم في مرحلة توليد الكود قد يتم حفظها في الجدول

SCANNER , LEXICAL ANALYZER
الفاحص –الماسح –المحلل الإملائي

SCANNER , LEXICAL ANALYZER



Interaction of lexical analyzer with parser

SCANNER , LEXICAL ANALYZER

Table for Examples of tokens

Token	**Sample Lexemes**	**Pattern**
const	const	const
If	if	if
Relation	<, <= ,= , < > ,>, >=	<or <= or = or < > or > or >=
Id	Pi , count ,D2	Letter followed by letters and digits
Num	3.1416,66, 2.02E-3	Any numeric constant
literal	“core dumped”	Any characters between “ and “ excepts “

SCANNER , LEXICAL ANALYZER
Attributes for Tokens

- Example: The tokens and associated attribute-values for the Fortran statement:

  E = M * C ** 2

- هذه العلامات والخواص هي لأحد التعبيرات

- Are written below as a sequence of pairs
- وهي يتم كتابتها كتتابعات في أزواج
- <**id** , pointer to symbol –table for E >
- <**assign-op**>
- < **id** , pointer to symbol –table for M >
- <**mult-op**>
- < **id** , pointer to symbol –table for C >
- <**exp--op**>

SCANNER , LEXICAL ANALYZE
Lexical Errors

- نلاحظ أن المصحح الإملائي له نظرة موضوعية جدا ومحدودة في البرنامج فمثلا لهذا التعبير fi (a = = f(x))
- يري المحلل الإملائي أن fi عبارة عن معرف identifier بالرغم من وضوح أنه خطأ key word “ if”
- وفي المترجمات الحديثة يتم تصحيح ذلك عن طريق
- Error recovery action
- تغطية وتصحيح الأخطاء البسيطة ومنها أيضا:
- Deleting an extraneous character
- Inserting a missing characters
- Replacing an incorrect character by correct character >

Specification of Tokens

- **الـ Prefix :-**
- **هو تتابع يتم الحصول عليه بإزالة Zero أو الحروف المتأخرة من التتابع مثال :**

- **Prefix (S) ← banana**
- **Prefix (S) ← banan**
- **Prefix (S) ← bana**
- **Prefix (S) ← ban**
- **Prefix (S) ← ba**

Specification of Tokens

- **الـ Suffix : -**
- **هو تتابع يتم الحصول عليه بإزالة Zero أو الحروف المتقدمة من التتابع مثال :**
- **Suffix (S) ← banana**
- **Suffix (S) ← anana**
- **Suffix (S) ← nana**
- **Suffix (S) ← ana**
- **Suffix (S) ← na**

Specification of Tokens

- **<u>الـ Sub String :-</u>**
- **تتابع يتم الحصول عليه من إزالة المقدمة والمؤخرة مثال :**
- **Sub String (S) ← anan**
- **Sub String (S) ← ana**
- **Sub String (S) ← nan**
- **Sub String (S) ← an**
- **Sub String (S) ← a**
- ***ملحوظة : معنى هذا أن كل Prefix و Suffix عبارة عن Sub string وليس العكس .**

Regular Definition التعريفات المنتظمة

- وهي تتم عندما نريد إعطاء اسماء للتعبيرات المنتظمة ثم تعريق هذه التعبيرات المنتظمة بواسطة استخدام تلك الأسماء مع اعتبار تلك الأسماء رموز
- فمثلا لو أن $\sum$ عبارة عن الحروف الهجائية الأساسية فتكون التعبيرات المنتظمة عبارة عن تتابعات من التعريفات:
- $d_1 \dashrightarrow r_1$ & $d_2 \dashrightarrow r_2$ $d_3 \dashrightarrow r_3$ & $d_n \dashrightarrow r_n$
- حيث d_i عبارة عن اسم وحيد
- وكل r_i تعبير منتظم علي الرموز $\sum U(d_1 ,d_2, ..., d_{i-1})$
- أي الرموز الأساسية وكذلك الأسماء التي تم تعريفها .... $d_1 ,d_2,$
- وهكذا يمكن تكوين تعبير منتظم علي الرموز الأساسية لأي وذلك بتكرار وإحلال اسماء تلك التعبيرات المنتظمة

Regular Definition التعريفات المنتظمة

- ولتمييز الأسماء من الرموز يمكن كتابة اسماء التعريفات المنتظمة ب Bold
- مثال :
- **letter** → A|B……|Z|a|b|…..|z
- **Digit** → 0|1|……|9
- **id → letter (letter | digit)***
- مثال : الأعداد في لغة الباسكال ولغة ال ++Unsigned numbers C
- مثل 1.6765E-45 , 6.67E4 , 39.45 , 5280
- ويمكن تمثيلها بالتعريفات المنتظمة كألأتي

Regular Definition التعريفات المنتظمة

- **digit** → 0|1|……|9
- **digits → digit digit***
- **Optional – fraction → . digits|** ε
- **Optional – exponent** → (E + |- | ε) **digits**) | ε
- **num→digits Optional–fraction Optional–exponent**

SCANNER , LEXICAL ANALYZE
Notional short hands

- قد نلجأ الي بعض الإختصارات المفيدة خاصة اذا بعض التركيبات للتعبيرات المنتظمة يتم استخدامها كطثيرا ويتكرار
- r^+ معناها بداية من الرمز نفسه(..... r , r^2)
- ? معناها لاشئ أو الموجود
- مثل $r?$ بديلا عن $r|\varepsilon$
- وهكذ نقول أن $r^* = r^+ ?$

SCANNER , LEXICAL ANALYZE
Notional short hands

- ويمكن كتابة الأعداد بصوره عامة كألتي:-

- **digit** → 0|1|……|9
- **digits → digit**$^{+}$
- **Optional – fraction → (. Digits**) ?
- **Optional–exponent →** (E (+ |-)?) **digits**)?
- **num→digits Optional–fraction Optional–exponent**
- → **digit**$^{+}$ **(. Digits**) ? (E (+ |-)?) **digits**)?

SCANNER , LEXICAL ANALYZE
Notional short hands

- بالنسبة للرموز characters
- |a +b +c| = a|b|c =|abc| = |a-c|
- |a-z| = a|b|c….|z
- وهكذا يمكن تعريف المعرفات identifier كالأتي:
- |A – Za –z | |A – Z a –z 0 -9 |*

Scanner- Lexical analyzer

خوارزم DFA

١– الدخل : هو عبارة عن أي كلمة مطلوب اختيارها .

٢– الخرج : الكلمة مقبولة أو غير مقبولة .

٣– البرنامج :

```
S:= So الحالة الابتدائية
C:= next char
while c != eof do
S:= MOVE ( S,C)
C:= next char ;
End ;
if S is in f return "Yes"
else return "No"
```