COMPILATEUR DE LANGAGE C

Presentés par:
El walid Abolaakoul
Ahmed Ben Ahmed

Encadré par:
MR.Abdeljalil SAKAT

HISTOIRE

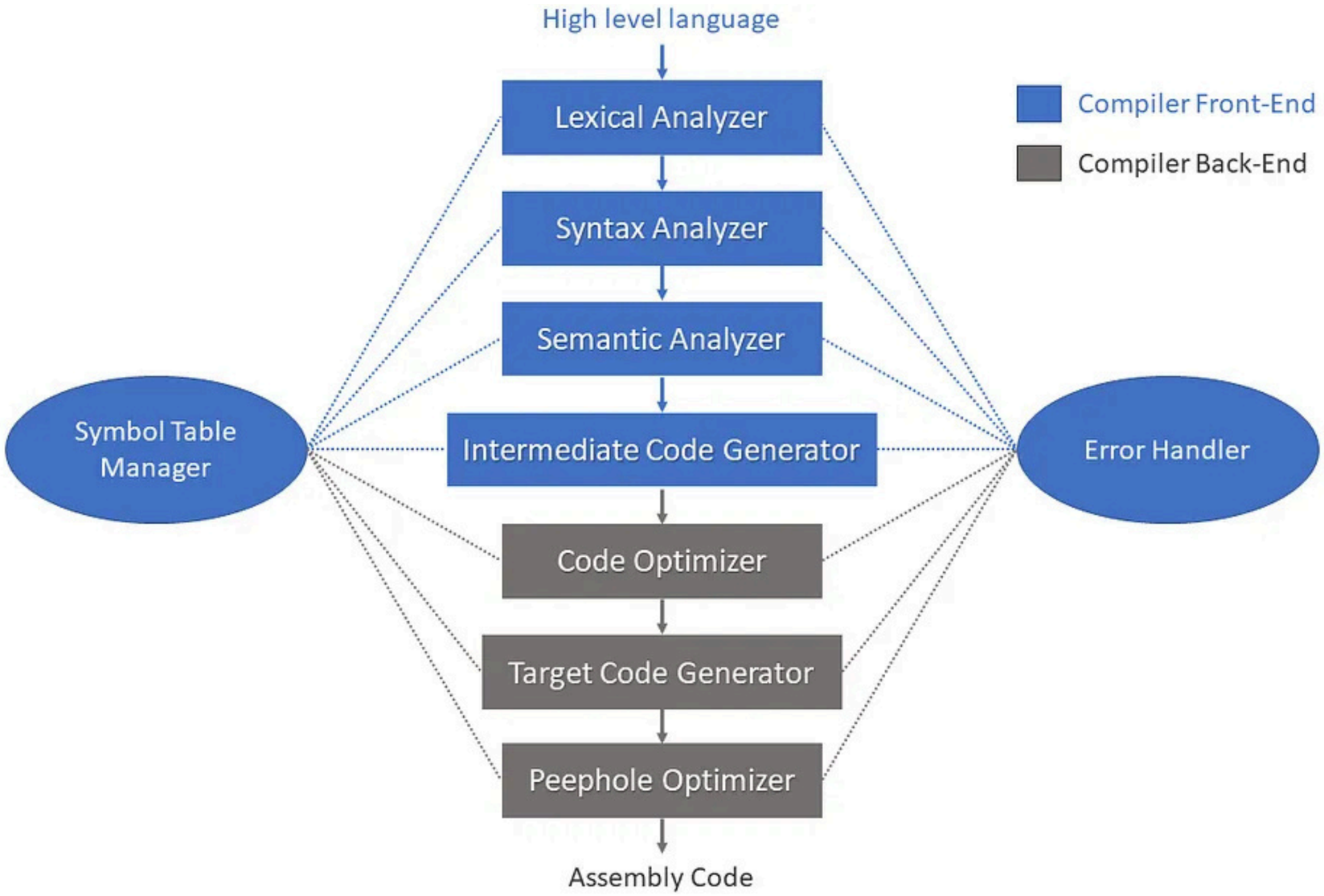
BOOTSTRAPPING DES COMPILATEURS

LES COMPILATEURS ÉTAIENT ENCORE RARES, ET LA PLUPART DES LOGICIELS ÉTAIENT ÉCRITS DIRECTEMENT EN CODE MACHINE.

1950 FORTRAN, 1970 C (OS UNIX), CPP, JAVA, PYTHON3.11

CRÉER UN PROGRAMME SIMPLE POUR EN FAIRE UN PROGRAMME PLUS COMPLEXE.

COMPILER STEPS



FRONT-END DU COMPILATEUR:

PRODUIT UN CODE INTERMÉDIAIRE (IR)

PEU IMPORTE LE LANGAGE D'ENTRÉE OU L'ARCHITECTURE CIBLE (EX : GCC SUPPORTE PLUSIEURS LANGAGES).:

LE CODE INTERMÉDIAIRE EST SIMPLE ET STANDARDISÉ,PLATFORME INDÉPENDANT, FACILITANT LES OPTIMISATIONS

IL SUFFIT DE RÉÉCRIRE LE BACK-END POUR CHAQUE MACHINE CIBLE(MAC OU PC), CE QUI ÉCONOMISE BEAUCOUP DE TRAVAIL

ROMINA JAFARIAN (TÉHÉRAN) A CRÉÉ UN INTERPRÉTEUR

EXÉCUTER DIRECTEMENT LE CODE INTERMÉDIAIRE À TROIS ADRESSES GÉNÉRÉ PAR LE FRONT-END

SANS PASSER PAR L'ASSEMBLAGE OU LE LINKING, FACILITANT AINSI LE TEST DU COMPILATEUR, DEBOGAGE

Exemples [modifier]

Dans le code à trois adresses, cela serait décomposé en plusieurs instructions distinctes. Ces instructions se traduisent plus facilement par langage d'assemblage. Il est également plus facile à détecter sous-expressions communes pour raccourcir le code. Dans l'exemple suivant, un calcul est composé de plusieurs plus petits:

```
# Calculer une solution à la [[équation
quadratique]].
x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)
```

```
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
```

LEXICAL ANALYSER

simple-c-compiler > input > C input_hello_world.c > main(void)

```
...
1  void main(void) {
2      output(1234);
3  }
```

```
(KEYWORD, void) (ID, main) (SYMBOL, () (KEYWORD, void) (SYMBOL, )) (SYMBOL, {)
(ID, output) (SYMBOL, () (NUM, 1234) (SYMBOL, )) (SYMBOL, ;)
(SYMBOL, })
```

Token type	Description
`KEYWORD`	Reserved keywords are essential to the program syntax and parsing. They are usually associated with many code generation and type checking steps. They can also be thought as hints for the compiler to produce certain intermediate code instructions in certain parts of the program. For example `void` keyword in our example denotes the type of the function or its arguments. This means they have no type i.e. the type doesn't exist, which in the typical programming notation translates to the function not returning anything or not accepting any arguments. It is essential for the compiler to know this to produce correct programs, hence these are reserved "keywords" that can't be used as variable or function names.
`ID`	An identifier which can be any name (except a `KEYWORD`) for a variable, argument or a function. These are stored in the symbol table (see figure above) as rows and different attributes are associated with them such as their address in memory or the type of the identifier (e.g it is a function, variable or function argument).
`NUM`	Numerical constant values (in our case integers) used in the computation. These are the very core of every program as playing with numbers and doing computation is essentially what every program does and variables typically need to be initialized with numbers before they can do anything useful (sometimes they're initialized with user input though). We could also have similar type for character string literals but they are not part of the implementation of our Simple C Compiler.
`SYMBOL`	Any special character, which is a part of the program syntax such as parentheses or brackets used to divide program into meaningful sections or computational units like functions or function argument lists.
`WHITESPACE`	It might be useful to have this token type as well for spaces, new lines, tabs etc. when implementing the lexical analyzer (for example to know where a word ends), but this type of token is just discarded in the end (as you can see from the example output). This is because the syntax of our programming language is not interested in how many whitespaces there are between words and symbols, they are only needed to separate different meaningful tokens.
`COMMENT`	This token is similar to `WHITESPACE` because it's also discarded from the token stream. It consists of the whole comment block or line (in case of line comment starting with `//`). This "token" type is somewhat counter intuitive as its lexim can consist of whole paragraphs of English sentences, but it's useful for implementing the lexical analyser.

DANS UN COMPILATEUR À PASSAGE UNIQUE ET EN PIPELINE, ON TRAITE UN TOKEN À LA FOIS, RÉDUISANT AINSI L'UTILISATION DE LA MÉMOIRE, MÊME AVEC DE GRANDS FICHIERS SOURCE.

DANS DES LANGAGES COMME LE C, L'ORDRE DES DÉCLARATIONS DE FONCTIONS EST IMPORTANT, UNE CONSÉQUENCE DE LA COMPILATION À PASSAGE UNIQUE

DANS UN COMPILATEUR À PASSAGE UNIQUE ET EN PIPELINE, ON TRAITE UN TOKEN À LA FOIS, RÉDUISANT AINSI L'UTILISATION DE LA MÉMOIRE, MÊME AVEC DE GRANDS FICHIERS SOURCE.

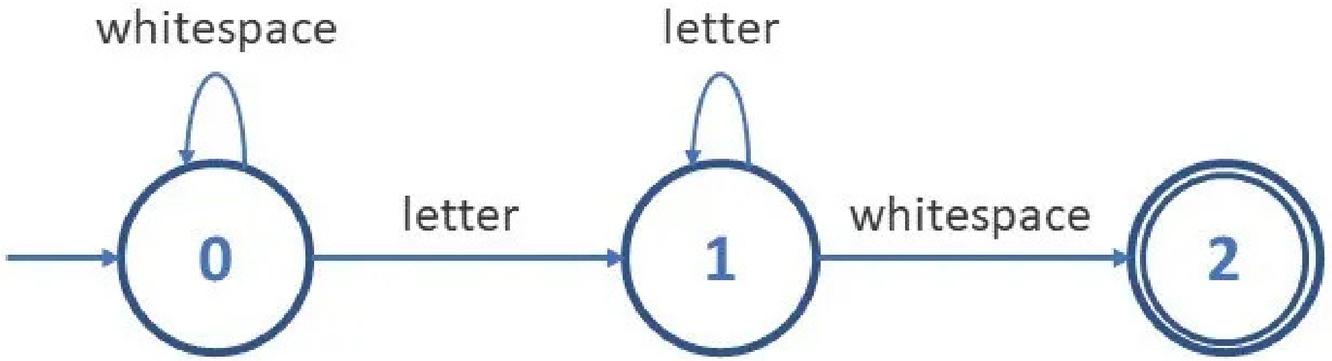
ALORS TOUT SIMPLEMENT ON TRAITE LE CODE POUR VERIFIER SON LEXIQUE

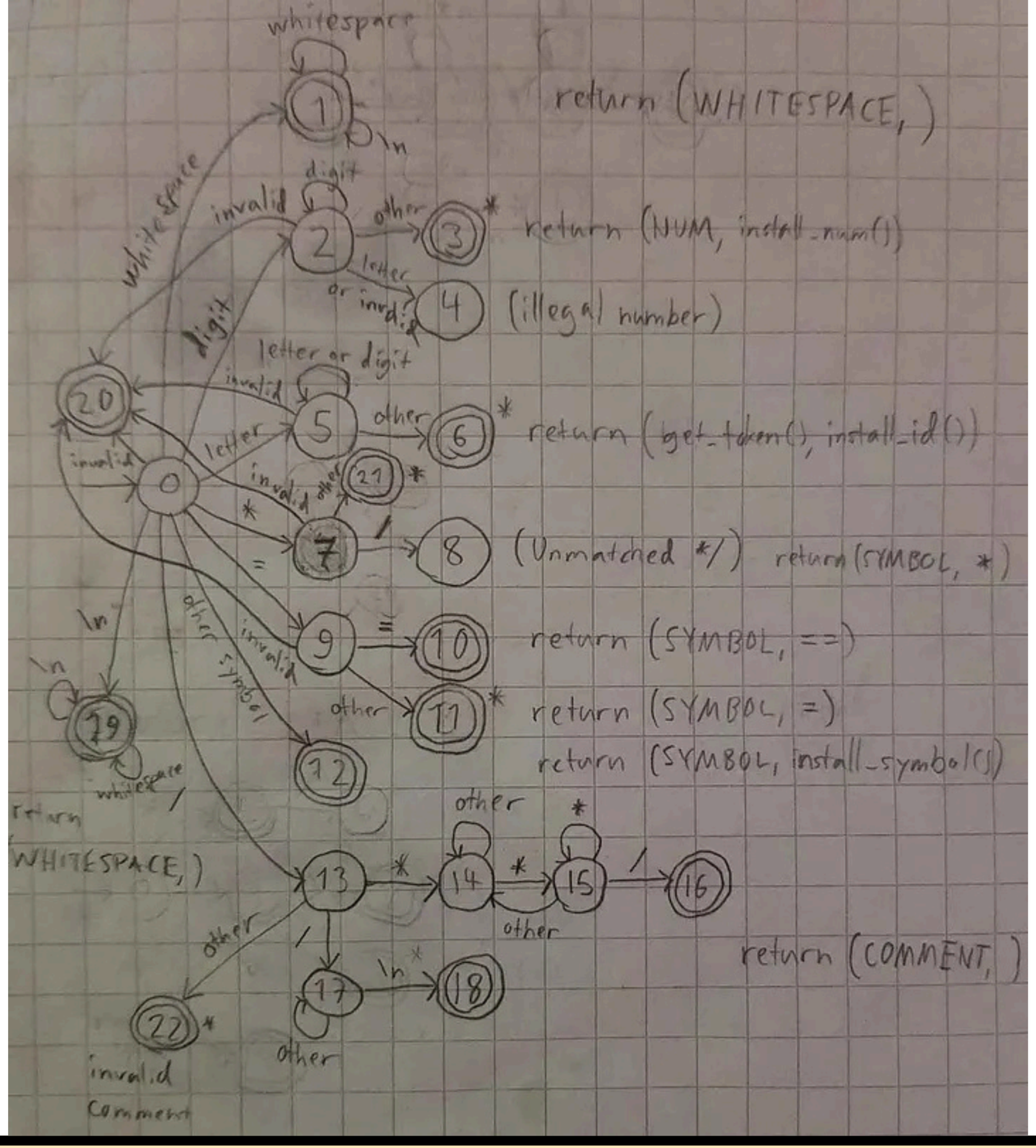
ET ON FAIT CELA TOKEN APRES TOKEN

ON RETOURNE LE RESULTAT DE CETTE ANALYSE HMMM

QUI PEUT FAIRE ÇA ???

AUTOMATE FINI DETERMINISTE




whitespace
1
return (WHITESPACE,)
\n
digit
invalid
whitespace
2
other
3
*
return (NUM, install_num())
letter
or invalid
4
(illegal number)
digit
letter or digit
invalid
20
5
other
6
*
return (get_token(), install_id())
invalid
letter
0
invalid
other
21
*
*
7
/
8
(Unmatched */)
return (SYMBOL, *)
=
9
=
10
return (SYMBOL, ==)
\n
other
invalid
other
11
*
return (SYMBOL, =)
return (SYMBOL, install_symbol())
\n
19
12
whitespace
/
return
(WHITESPACE,)
other
*
13
*
14
*
15
/
16
other
other
/
return (COMMENT,)
\n
*
17
18
other
22
*
invalid
comment

```
}

token_dfa = (
    # Input character types
    #   w     d     l     *     =     s     /    \n     o
    #   0     1     2     3     4     5     6     7     8
    (   1,    2,    5,    7,    9,   12,   13,   19,   20), # State 0 (initial
    (   1, None, None, None, None, None, None,    1, None), # State 1 (whitespa
    (   3,    2,    4,    3,    3,    3,    3,    3,    4), # State 2
    (None, None, None, None, None, None, None, None, None), # State 3 (number)
    (None, None, None, None, None, None, None, None, None), # State 4 (illegal
    (   6,    5,    5,    6,    6,    6,    6,    6,   20), # State 5
    (None, None, None, None, None, None, None, None, None), # State 6 (id or ke
    (  21,   21,   21,   21,   21,   21,    8,   21,   20), # State 7
    (None, None, None, None, None, None, None, None, None), # State 8 (unmatche
    (  11,   11,   11,   11,   10,   11,   11,   11,   20), # State 9
    (None, None, None, None, None, None, None, None, None), # State 10 (symbol
    (None, None, None, None, None, None, None, None, None), # State 11 (symbol
    (None, None, None, None, None, None, None, None, None), # State 12 (symbol)
    (  22,   22,   22,   14,   22,   22,   17,   22,   22), # State 13
    (  14,   14,   14,   15,   14,   14,   14,   14,   14), # State 14
    (  14,   14,   14,   15,   14,   14,   16,   14,   14), # State 15
    (None, None, None, None, None, None, None, None, None), # State 16 (/* comm
    (  17,   17,   17,   17,   17,   17,   17,   18,   17), # PState 17
    (None, None, None, None, None, None, None, None, None), # State 18 (// comm
    (  19, None, None, None, None, None, None,   19, None), # State 19 (newline
    (None, None, None, None, None, None, None, None, None), # State 20 (invalid
    (None, None, None, None, None, None, None, None, None), # State 21 (symbol
    (None, None, None, None, None, None, None, None, None), # State 22 (invalid
)

F = {1, 3, 6, 10, 11, 12, 16, 18, 19, 20, 21} # all accepting states
Fstar = {3, 6, 11, 21}                        # accepting states that require t
```

L'ANALYSE SYNTAXIQUE

Le parseur : une étape clé dans les compilateurs.

Transforme les tokens (produits par l'analyse lexicale) en une structure compréhensible.

```
if (a < b) {
    return 2 + 2;
}
```



```
(KEYWORD, if) (SYMBOL, () (ID, a) (SYMBOL, <) (ID, b) (SYMBOL, )) (SYMBOL, {)
(KEYWORD, return) (NUM, 2) (SYMBOL, +) (NUM, 2) (SYMBOL, ;)
```

L'analyse syntaxique est une étape essentielle dans les compilateurs. Elle permet de vérifier si le programme respecte les règles de la grammaire. Le parseur transforme les tokens, fournis par l'analyse lexicale, en une structure logique.

L'ANALYSE SYNTAXIQUE

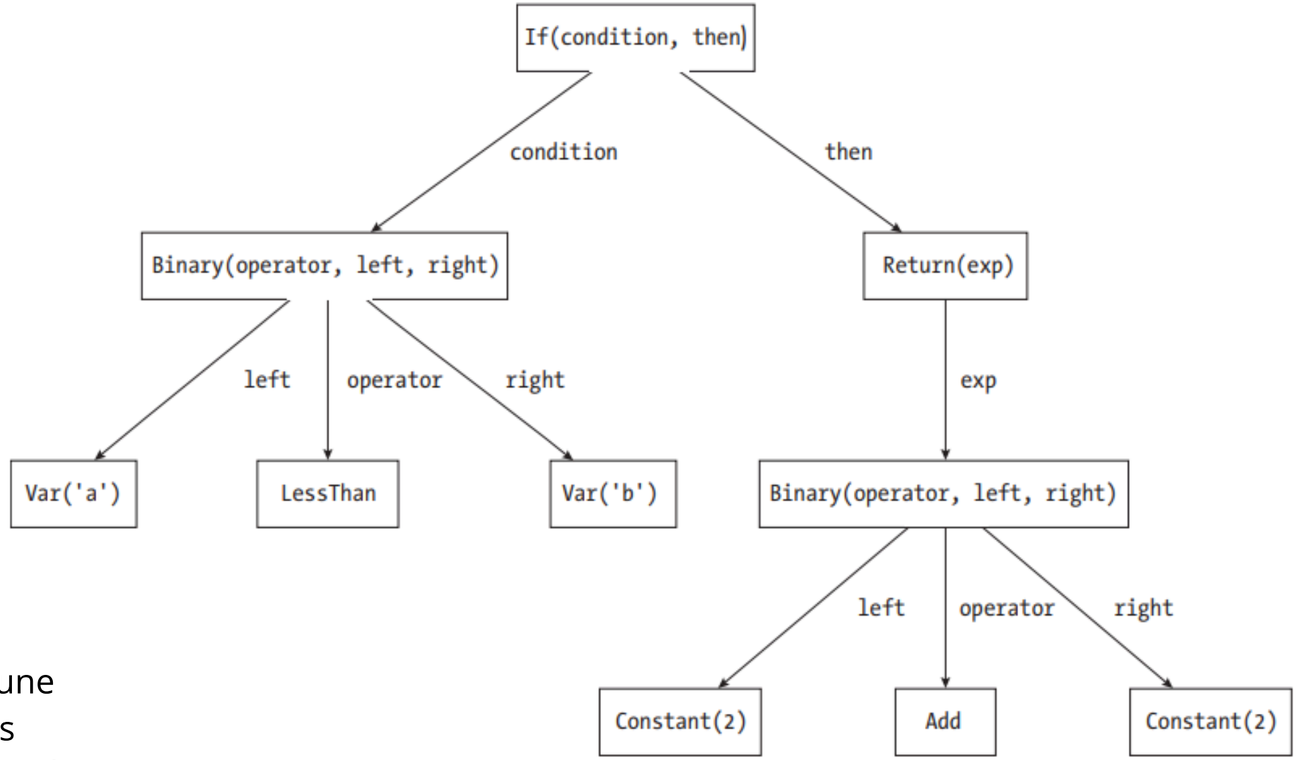
- Il construit une structure appelée arbre de syntaxe.
- Utilise une grammaire pour comprendre le programme.
- Identifie les erreurs syntaxiques.

L'analyse syntaxique est une étape essentielle dans les compilateurs. Elle permet de vérifier si le programme respecte les règles de la grammaire. Le parseur transforme les tokens, fournis par l'analyse lexicale, en une structure logique.

L'ANALYSE SYNTAXIQUE

Il construit une structure appelée arbre de syntaxe.

```
if (a < b) {
    return 2 + 2;
}
```

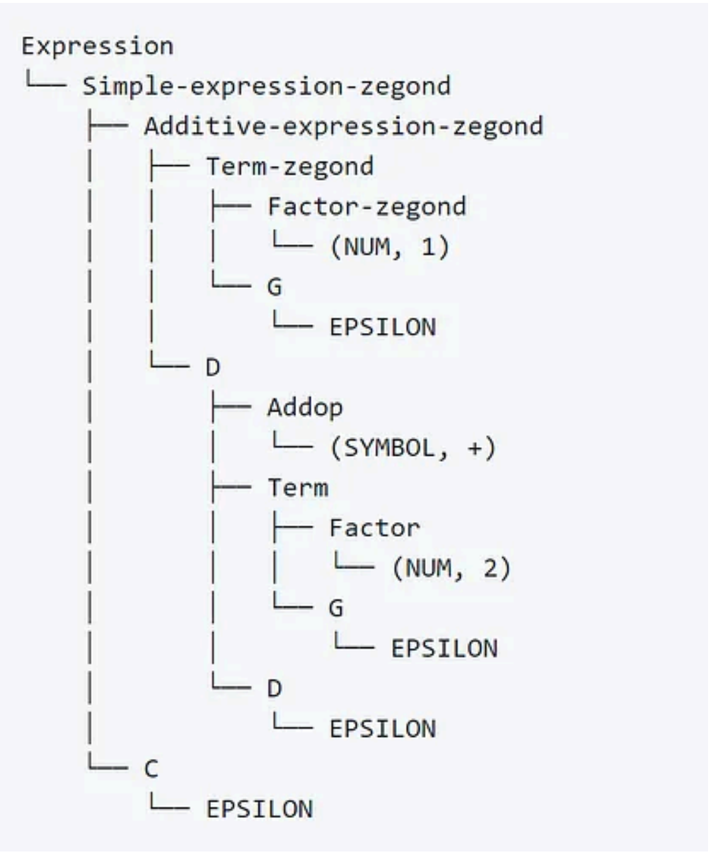


Voici un exemple d'arbre de syntaxe. Chaque noeud représente une partie importante de la structure, comme les opérateurs ou les variables. Les feuilles, elles, représentent les éléments élémentaires de notre programme.

L'ANALYSE SYNTAXIQUE

Il construit une structure appelée arbre de syntaxe.

```
void main(void) {
    output(1 + 2);
}
```



Voici un exemple d'arbre de syntaxe. Chaque noeud représente une partie importante de la structure, comme les opérateurs ou les variables. Les feuilles, elles, représentent les éléments élémentaires de notre programme.

L'ANALYSE SYNTAXIQUE

Utilise une grammaire pour comprendre le programme.

```
if (a < b) {
    return 2 + 2;
}
```


<programme> ::= <instruction>

<instruction> ::= "if" "(" <expression> ")" "{" <instruction> "}" | "return" <expression> ";”

<expression> ::= <terme> | <terme> <opérateur> <terme>

<terme> ::= <identifiant> | <entier>

Voici un exemple d’arbre de syntaxe. Chaque noeud représente une partie importante de la structure, comme les opérateurs ou les variables. Les feuilles, elles, représentent les éléments élémentaires de notre programme.

L'ANALYSE SYNTAXIQUE

Supposons que nous ayons cette grammaire LL(1)...

```
1-3     S → A ; | for ( A ; C ; A ) S | B
4-5     A → V = E | ε
6-7     C → E | ε
8       E → V
9       V → id X
10-11   X → and V | ε
12      B → { L }
13-14   L → S L | ε
```

tokens coming from the scanner

	;	id	and	for	)	{	}	=	$
S	A ;	A ;		for (A ; C ; A) S		B			synch
A	ε	V = E			ε				ε
C	ε	E							ε
E	synch	V			synch				synch
V	synch	id X			synch			synch	synch
X	ε		and V		ε			ε	ε
B	synch	synch		synch		{ L }			synch
L	S L	S L		S L		S L	ε		ε

the non-terminal lying on the top of the stack of the LL(1)

Une grammaire LL(1) est un type particulier de grammaire hors-contexte qui peut être encodée de manière non ambiguë dans une table de parsing en deux dimensions, tout en ne regardant qu'un seul jeton à la fois (exactement comme nous le souhaitons !).

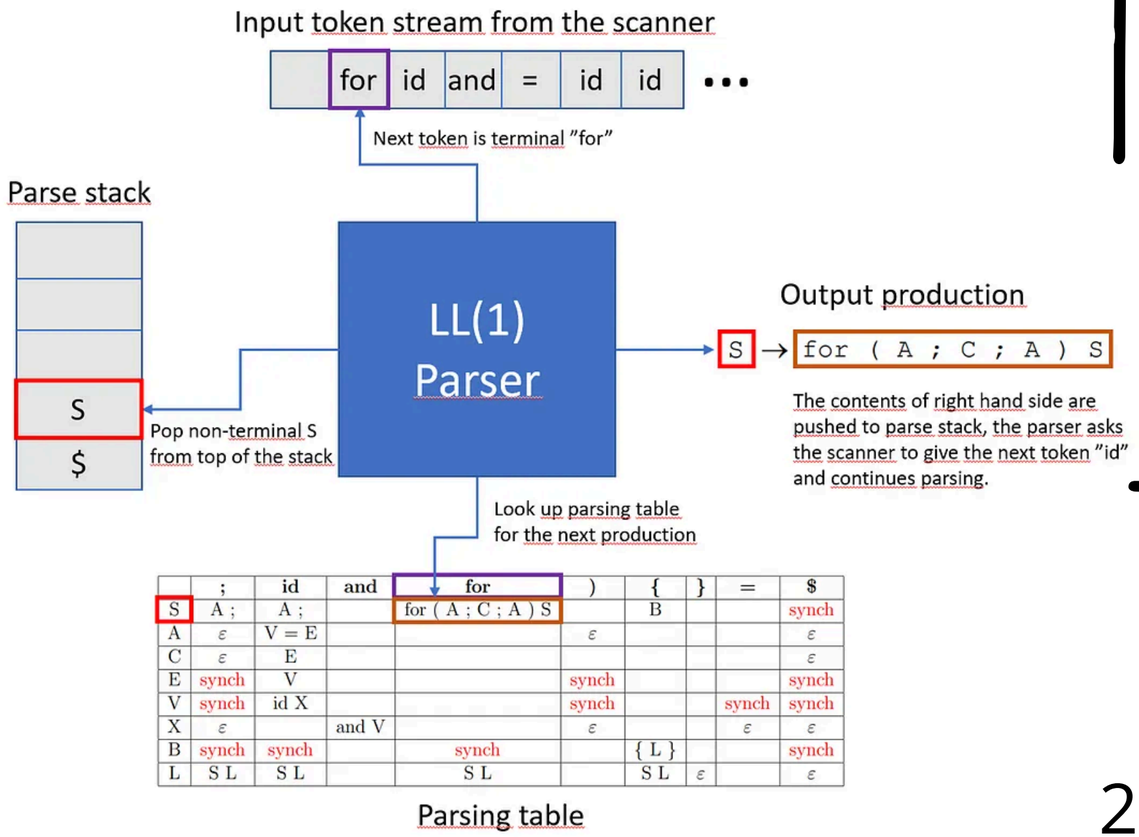
Synch ???

En parsing, synch est utilisé pour la récupération des erreurs, notamment dans les algorithmes de parsing LL(1).
Cela améliore la robustesse du parseur en lui permettant de gérer plusieurs erreurs sans arrêter l'exécution.

L'ANALYSE SYNTAXIQUE

Prenons l'exemple: for id and = id id ; ;) ; $

Stack	Input	Action	Error
S $	for id and = id id ; ;) ; $	for (A ; C ; A) S	
for (A ; C ; A) S $	for id and = id id ; ;) ; $	terminal	
(A ; C ; A) S $	id and = id id ; ;) ; $	pop (	Missing '('
A ; C ; A) S $	id and = id id ; ;) ; $	V = E	
V = E ; C ; A) S $	id and = id id ; ;) ; $	id X	
id X = E ; C ; A) S $	id and = id id ; ;) ; $	terminal	
id X = E ; C ; A) S $	id and = id id ; ;) ; $	terminal	
X = E ; C ; A) S $	and = id id ; ;) ; $	and V	
and V = E ; C ; A) S $	and = id id ; ;) ; $	terminal	
V = E ; C ; A) S $	= id id ; ;) ; $	pop V	Missing Term
= E ; C ; A) S $	= id id ; ;) ; $	terminal	
E ; C ; A) S $	id id ; ;) ; $	V	
V ; C ; A) S $	id id ; ;) ; $	id X	
id X ; C ; A) S $	id id ; ;) ; $	terminal	
X ; C ; A) S $	id ; ;) ; $	skip id	Misplaced id
X ; C ; A) S $	; ;) ; $	ε	
; C ; A) S $	; ;) ; $	terminal	
C ; A) S $	;) ; $	ε	
; A) S $	;) ; $	terminal	
A) S $	) ; $	ε	
) S $	) ; $	terminal	
S $	; $	A ;	
A ; $	; $	ε	
; $	; $	terminal	
$	$	HALT	

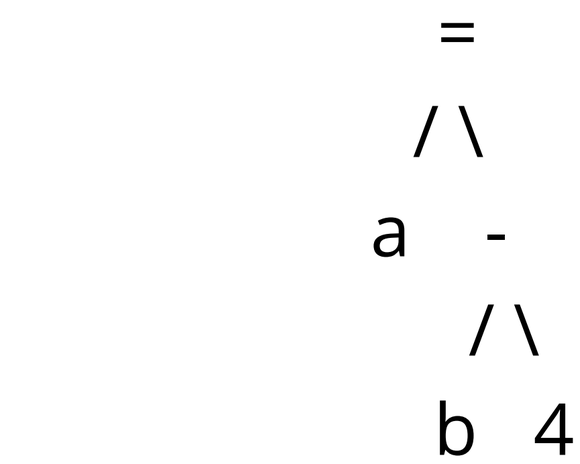


	;	id	and	for	)	{	}	=	$
S	A ;	A ;		for (A ; C ; A) S		B			synch
A	ε	V = E			ε				ε
C	ε	E							ε
E	synch	V			synch				synch
V	synch	id X			synch			synch	synch
X	ε		and V		ε			ε	ε
B	synch	synch		synch		{ L }			synch
L	S L	S L		S L		S L	ε		ε

Parsing table

L'ANALYSE SYNTAXIQUE

À ce stade, un arbre syntaxique abstrait (AST) est génér



ANALYSE SÉMANTIQUE

VÉRIFIE SI LE PROGRAMME EST LOGIQUE ET COHÉRENT DANS SON CONTEXTE.

LE LEXER ET LE PARSER S'OCCUPENT DE LA STRUCTURE SYNTAXIQUE DES PROGRAMMES, L'ANALYSEUR SÉMANTIQUE VÉRIFIE SI CES PROGRAMMES ONT UN SENS

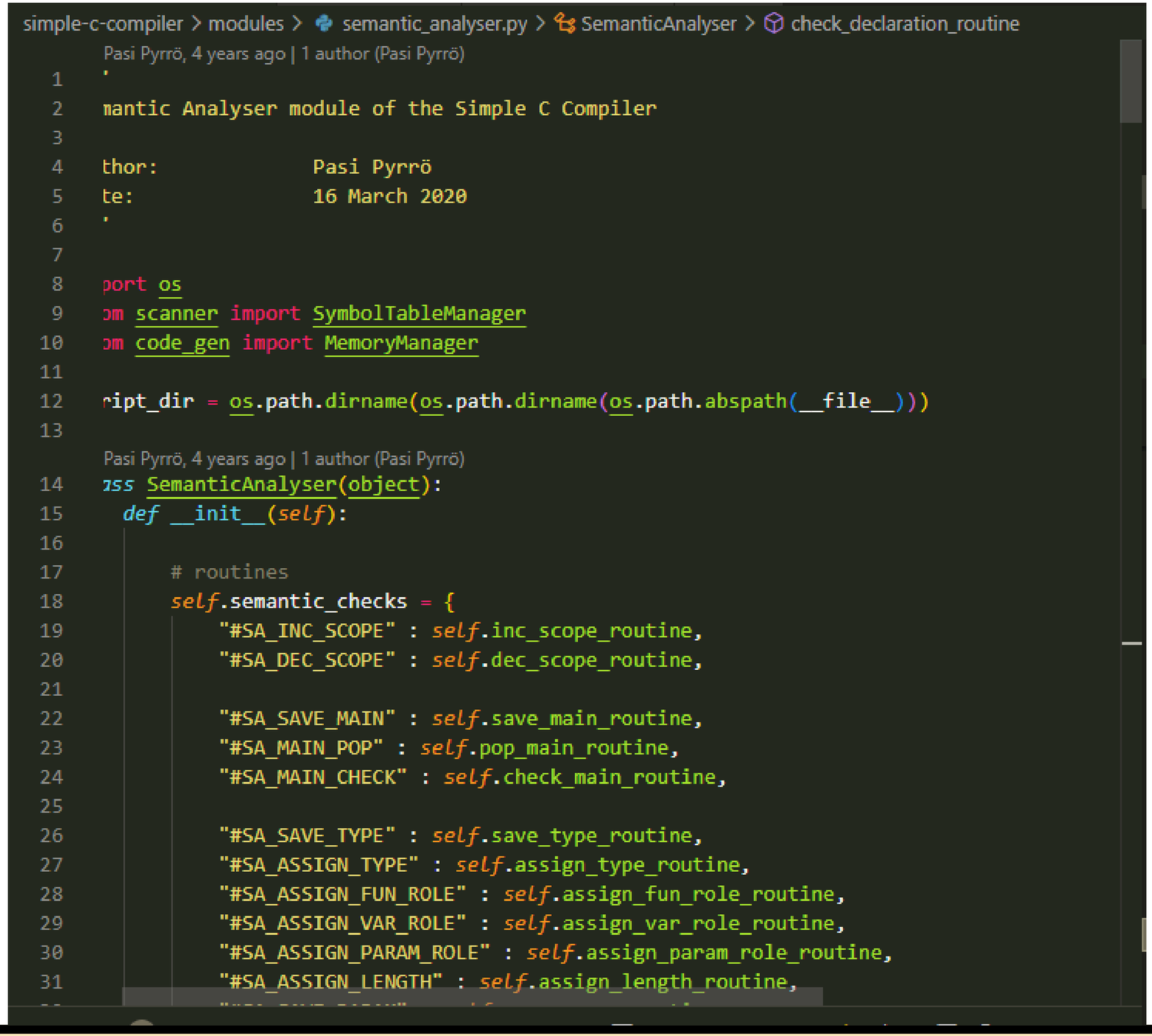
LES RÈGLES DU LANGAGE AU-DELÀ DE LA GRAMMAIRE ET DÉTECTE DES ERREURS QUI NE PEUVENT PAS ÊTRE IDENTIFIÉES PAR L'ANALYSEUR SYNTAXIQUE SEUL,

- **LES ERREURS LIÉES AU CONTEXTE.**
- **LES INCOHÉRENCES DE TYPE.**
- **L'UTILISATION INCORRECTE DE CONSTRUCTIONS DU LANGAGE.**

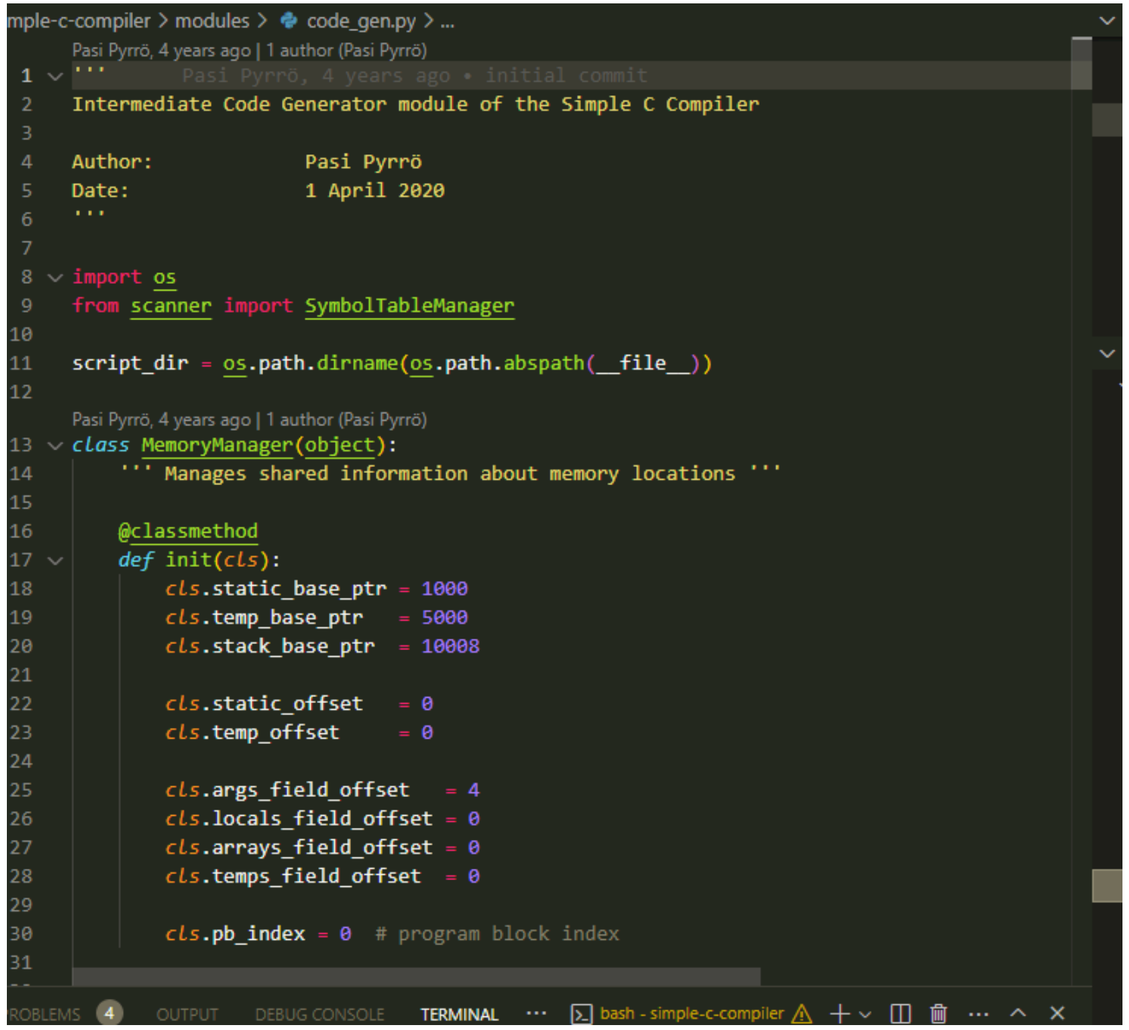
SORTIE DE L'ANALYSEUR SÉMANTIQUE
IDÉALEMENT : AUCUNE SORTIE !

EN CAS D'ERREURS, ELLES SONT SIGNALÉES SOUS FORME DE MESSAGES EXPLICITES.

- **MAIN MANQUANTE : UNE FONCTION VOID MAIN(VOID) EST OBLIGATOIRE POUR EXÉCUTER UN PROGRAMME.**
- **DÉCLARATION OBLIGATOIRE : LES VARIABLES DOIVENT ÊTRE DÉCLARÉES AVANT UTILISATION DANS LA PORTÉE COURANTE.**
- **TYPE VOID INTERDIT : LES VARIABLES NE PEUVENT PAS AVOIR LE TYPE VOID.**
- **ARGUMENTS DE FONCTION : VÉRIFIEZ LE NOMBRE ET LE TYPE DES ARGUMENTS PASSÉS À UNE FONCTION.**
- **BREAK ET CONTINUE : CES INSTRUCTIONS NE PEUVENT ÊTRE UTILISÉES QU'À L'INTÉRIEUR DES BOUCLES.**
- **TYPES COMPATIBLES : LES OPÉRATIONS BINAIRES NÉCESSITENT DES TYPES COMPATIBLES POUR ÉVITER LES ERREURS.**
- **TABLE DES SYMBOLES : ELLE EST MISE À JOUR AVEC LES TYPES ET LES DÉCLARATIONS POUR CHAQUE VARIABLE.**
- **ERREURS SIGNALÉES : L'ANALYSEUR NE GÉNÈRE PAS DE SORTIE, SAUF POUR SIGNALER DES ERREURS.**
- **VALIDATION DES CONTEXTES : VÉRIFIE QUE CHAQUE INSTRUCTION EST UTILISÉE DANS UN CONTEXTE APPROPRIÉ.**
- **INFÉRENCE DE TYPES : L'ANALYSEUR DÉDUIT LES TYPES DES EXPRESSIONS ET DES IDENTIFICATEURS.**


simple-c-compiler > modules > semantic_analyser.py > SemanticAnalyser > check_declaration_routine
Pasi Pyrrö, 4 years ago | 1 author (Pasi Pyrrö)
'
mantic Analyser module of the Simple C Compiler
thor: Pasi Pyrrö
te: 16 March 2020
'
port os
m scanner import SymbolTableManager
m code_gen import MemoryManager
ript_dir = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
Pasi Pyrrö, 4 years ago | 1 author (Pasi Pyrrö)
ss SemanticAnalyser(object):
def __init__(self):
# routines
self.semantic_checks = {
"#SA_INC_SCOPE" : self.inc_scope_routine,
"#SA_DEC_SCOPE" : self.dec_scope_routine,
"#SA_SAVE_MAIN" : self.save_main_routine,
"#SA_MAIN_POP" : self.pop_main_routine,
"#SA_MAIN_CHECK" : self.check_main_routine,
"#SA_SAVE_TYPE" : self.save_type_routine,
"#SA_ASSIGN_TYPE" : self.assign_type_routine,
"#SA_ASSIGN_FUN_ROLE" : self.assign_fun_role_routine,
"#SA_ASSIGN_VAR_ROLE" : self.assign_var_role_routine,
"#SA_ASSIGN_PARAM_ROLE" : self.assign_param_role_routine,
"#SA_ASSIGN_LENGTH" : self.assign_length_routine,

CODE GEN


mple-c-compiler > modules > code_gen.py > ...
Pasi Pyrrö, 4 years ago | 1 author (Pasi Pyrrö)
''' Pasi Pyrrö, 4 years ago • initial commit
Intermediate Code Generator module of the Simple C Compiler
Author: Pasi Pyrrö
Date: 1 April 2020
'''
import os
from scanner import SymbolTableManager
script_dir = os.path.dirname(os.path.abspath(__file__))
Pasi Pyrrö, 4 years ago | 1 author (Pasi Pyrrö)
class MemoryManager(object):
''' Manages shared information about memory locations '''
@classmethod
def init(cls):
cls.static_base_ptr = 1000
cls.temp_base_ptr = 5000
cls.stack_base_ptr = 10008
cls.static_offset = 0
cls.temp_offset = 0
cls.args_field_offset = 4
cls.locals_field_offset = 0
cls.arrays_field_offset = 0
cls.temps_field_offset = 0
cls.pb_index = 0 # program block index
ROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL bash - simple-c-compiler

- **INITIALISATION DE LA MÉMOIRE : ASSIGNER DES VALEURS CONSTANTES À DES ADRESSES MÉMOIRE POUR STOCKER LES RÉSULTATS.**

- **OPÉRATION D'ADDITION : TRADUIRE L'ADDITION EN INSTRUCTION INTERMÉDIAIRE (ADD, #1, #2, 5000).**

- **VALIDATION DES TYPES : VÉRIFIER LA COMPATIBILITÉ DES TYPES DES OPÉRANDES AVANT DE GÉNÉRER LE CODE.**

- **UTILISATION DE LA MÉMOIRE : STOCKER LE RÉSULTAT DE L'ADDITION DANS UNE ADRESSE MÉMOIRE (EX. 5000).**

- **OPTIMISATION À POSTERIORI : SIMPLIFIER LE CODE GÉNÉRÉ EN RÉDUISANT LE NOMBRE D'INSTRUCTIONS.**

- **STOCKAGE DES RÉSULTATS : ASSOCIER CHAQUE CALCUL À UNE ADRESSE MÉMOIRE POUR UNE RÉCUPÉRATION FUTURE.**

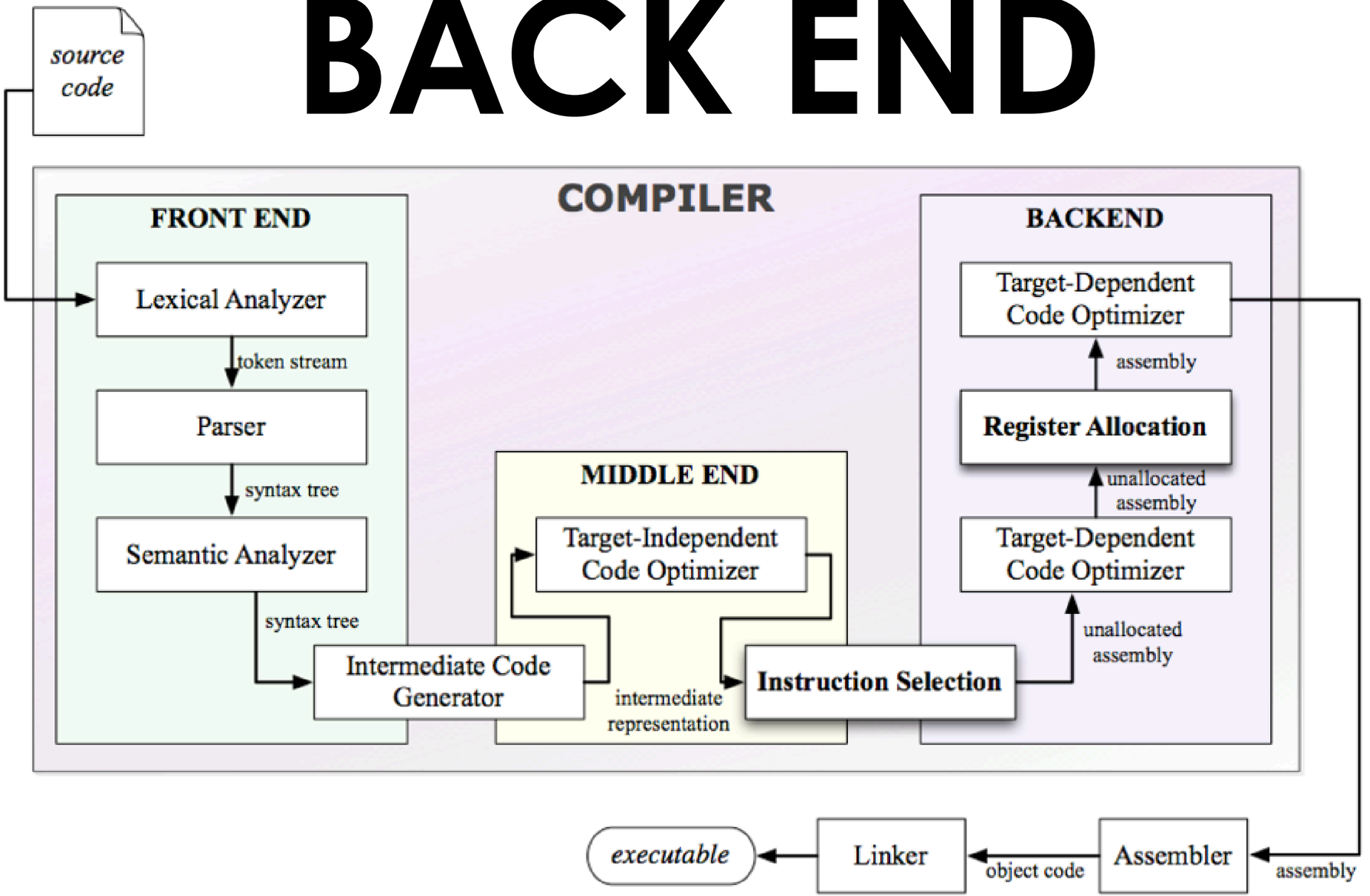
- **SÉCURITÉ DES VALEURS : VÉRIFIER QUE LES VARIABLES SONT CORRECTEMENT TYPÉES AVANT LE TRAITEMENT.**

- **GÉNÉRATION DE TROIS ADRESSES : UTILISER UNE STRUCTURE À TROIS ADRESSES POUR CHAQUE OPÉRATION.**

- **VÉRIFICATION DES ERREURS : IDENTIFIER LES ERREURS COMME LES TYPES INCOMPATIBLES AVANT LA GÉNÉRATION DU CODE.**

- **TRANSITION VERS LE CODE MACHINE : CONVERTIR LE CODE INTERMÉDIAIRE EN INSTRUCTIONS MACHINE APRÈS VALIDATION.**

BACK END


source code
COMPILER
FRONT END
Lexical Analyzer
token stream
Parser
syntax tree
Semantic Analyzer
syntax tree
Intermediate Code Generator
MIDDLE END
Target-Independent Code Optimizer
intermediate representation
Instruction Selection
BACKEND
unallocated assembly
Target-Dependent Code Optimizer
Register Allocation
assembly
Target-Dependent Code Optimizer
unallocated assembly
assembly
Assembler
object code
Linker
executable

- LE BACK-END DU COMPILATEUR TRADUIT LE CODE INTERMÉDIAIRE EN CODE MACHINE SPÉCIFIQUE À L'ARCHITECTURE CIBLE.

- L'ALLOCATION DE REGISTRES DANS LE BACK-END RÉSOUT UN PROBLÈME DE COLORIAGE DE GRAPHES POUR OPTIMISER L'UTILISATION DES REGISTRES.

- LE CODE INTERMÉDIAIRE OPTIMISÉ EST TRANSFORMÉ EN CODE ASSEMBLEUR CIBLE AVEC UN PROCESSUS DE TABLE DE CORRESPONDANCE.

- LE PROCESSUS DE TRADUCTION EN ASSEMBLEUR EST UNE SIMPLE RECHERCHE DANS UNE TABLE DE CORRESPONDANCES.

- L’OPTIMISATION PEEPHOLE AMÉLIORE LES INSTRUCTIONS ASSEMBLEUR POUR RENDRE LE CODE PLUS EFFICACE SUR LE MATÉRIEL CIBLE.

- L’ASSEMBLEUR ET LE LINKER GÉNÈRENT UN FICHIER BINAIRE EXÉCUTABLE À PARTIR DU CODE MACHINE.

- LE BACK-END NÉCESSITE PLUSIEURS ÉTAPES DÉTAILLÉES POUR PRODUIRE UN BINAIRE, DÉPENDANTES DE LA PLATEFORME CIBLE.