ConvCC-2021-1

A seguir é apresentada a Gramática ConvCC-2021-1, onde sequências em maiúsculo e preto representam símbolos não-terminais, enquanto sequências em minúsculo e em vermelho repsentam símbolos terminais.

PROGRAM	→	STATEMENT | FUNCLIST | ε
FUNCLIST	→	FUNCDEF FUNCLIST | FUNCDEF
FUNCDEF	→	def ident(PARAMLIST) { STATELIST }
PARAMLIST	→	int ident, PARAMLIST | float ident, PARAMLIST | string ident, PARAMLIST | int ident | float ident | string ident | ε
STATEMENT	→	VARDECL; | ATRIBSTAT; | PRINTSTAT; | READSTAT; | RETURNSTAT; | IFSTAT | FORSTAT | {STATELIST} | break; | ;
VARDECL	→	int ident VARDECL' | float ident VARDECL' | string ident VARDECL'
VARDECL'	→	[int_constant] VARDECL' | ε
ATRIBSTAT	→	LVALUE = EXPRESSION | LVALUE = ALLOCEXPRESSION | LVALUE = FUNCCALL
FUNCCALL	→	ident(PARAMLISTCALL)
PARAMLISTCALL	→	ident, PARAMLISTCALL | ident | ε
PRINTSTAT	→	print EXPRESSION
READSTAT	→	read LVALUE
RETURNSTAT	→	return
IFSTAT	→	if (EXPRESSION) STATEMENT else STATEMENT | if (EXPRESSION) STATEMENT
FORSTAT	→	for(ATRIBSTAT; EXPRESSION; ATRIBSTAT) STATEMENT
STATELIST	→	STATEMENT | STATEMENT STATELIST
ALLOCEXPRESSION	→	new int ALLOCEXPRESSION' | new float ALLOCEXPRESSION' | new string ALLOCEXPRESSION'
ALLOCEXPRESSION'	→	[NUMEXPRESSION] | [NUMEXPRESSION] ALLOCEXPRESSION'
EXPRESSION	→	NUMEXPRESSION | NUMEXPRESSION < NUMEXPRESSION | NUMEXPRESSION > NUMEXPRESSION | NUMEXPRESSION <= NUMEXPRESSION | NUMEXPRESSION >= NUMEXPRESSION | NUMEXPRESSION == NUMEXPRESSION | NUMEXPRESSION != NUMEXPRESSION

NUMEXPRESSION	→	TERM NUMEXPRESSION'
NUMEXPRESSION'	→	+ TERM NUMEXPRESSION' | - TERM NUMEXPRESSION' | ε
TERM	→	UNARYEXPR TERM'
TERM'	→	* UNARYEXPR TERM' | / UNARYEXPR TERM' | % UNARYEXPR TERM' | ε
UNARYEXPR	→	+ FACTOR | - FACTOR | FACTOR
FACTOR	→	int_constant | float_constant | string_constant | null | LVALUE | (NUMEXPRESSION)
LVALUE	→	ident LVALUE'
LVALUE'	→	[NUMEXPRESSION] LVALUE' | ε

Recursão à esquerda

A gramática ConvCC-2021-1 não é recursiva à esquerda. Não existe nenhuma produção que, em 1 ou mais passos de derivação, gera uma produção do tipo `A → Aα`. Uma boa parte das produções começam produzindo símbolos não-terminais, o que facilita a análise.

Fatoração à esquerda

Na gramática apresentada acima, existem produções que não estão fatoradas à esquerda. São elas:

- `FUNCLIST`
- `PARAMLIST`
- `ATRIBSTAT`
- `PARAMLISTCALL`
- `IFSTAT`
- `STATELIST`
- `ALLOCEXPRESSION`
- `ALLOCEXPRESSION'`
- `EXPRESSION`

A seguir é apresentada a gramática fatorada à esquerda:

PROGRAM	→	STATEMENT | FUNCLIST | ε
FUNCLIST	→	FUNCDEF FUNCLIST'
FUNCLIST'	→	FUNCLIST | ε
FUNCDEF	→	def ident(PARAMLIST) { STATELIST }
PARAMLIST	→	int ident PARAMLIST' | float ident PARAMLIST' | string ident PARAMLIST' | ε
PARAMLIST'	→	, PARAMLIST | ε

STATEMENT	→	VARDECL; | ATRIBSTAT; | PRINTSTAT; | READSTAT; | RETURNSTAT; | IFSTAT | FORSTAT | {STATELIST} | break; | ;
VARDECL	→	int ident VARDECL' | float ident VARDECL' | string ident VARDECL'
VARDECL'	→	[int_constant] VARDECL' | ε
ATRIBSTAT	→	LVALUE = ATRIBSTAT'
ATRIBSTAT'	→	EXPRESSION | ALLOCEXPRESSION | FUNCCALL
FUNCCALL	→	ident(PARAMLISTCALL)
PARAMLISTCALL	→	ident PARAMLISTCALL' | ε
PARAMLISTCALL'	→	, PARAMLISTCALL | ε
PRINTSTAT	→	print EXPRESSION
READSTAT	→	read LVALUE
RETURNSTAT	→	return
IFSTAT	→	if (EXPRESSION) STATEMENT IFSTAT'
IFSTAT'	→	else STATEMENT | ε
FORSTAT	→	for(ATRIBSTAT; EXPRESSION; ATRIBSTAT) STATEMENT
STATELIST	→	STATEMENT STATELIST'
STATELIST'	→	STATELIST | ε
ALLOCEXPRESSION	→	new ALLOCEXPRESSION'
ALLOCEXPRESSION'	→	int ALLOCEXPRESSION'' | float ALLOCEXPRESSION'' | string ALLOCEXPRESSION''
ALLOCEXPRESSION''	→	[NUMEXPRESSION] ALLOCEXPRESSION'''
ALLOCEXPRESSION'''	→	ALLOCEXPRESSION'' | ε
EXPRESSION	→	NUMEXPRESSION EXPRESSION'
EXPRESSION'	→	< EXPRESSION'' | > EXPRESSION'' | == NUMEXPRESSION | != NUMEXPRESSION | ε
EXPRESSION''	→	NUMEXPRESSION | = NUMEXPRESSION
NUMEXPRESSION	→	TERM NUMEXPRESSION'
NUMEXPRESSION'	→	+ TERM NUMEXPRESSION' | - TERM NUMEXPRESSION' | ε
TERM	→	UNARYEXPR TERM'
TERM'	→	* UNARYEXPR TERM' | / UNARYEXPR TERM' | % UNARYEXPR TERM' | ε
UNARYEXPR	→	+ FACTOR | - FACTOR | FACTOR
FACTOR	→	int_constant | float_constant | string_constant | null | LVALUE | (NUMEXPRESSION)
LVALUE	→	ident LVALUE'

LVALUE'	→	[NUMEXPRESSION] LVALUE' | ε

Foram incluidos os símbolos não terminais `FUNCLIST'` `PARAMLIST'`, `ATRIBSTAT'`, `PARAMLISTCALL'`, `IFSTAT'`, `STATELIST'`, `ALLOCEXPRESSION''`, `ALLOCEXPRESSION'''`, `EXPRESSION'` e `EXPRESSION''`. Além disso, as produções de `ALLOCEXPRESSION'` foram alteradas.

LL(1)

Para determinar se a gramática é LL(1), vamos precisar de todos os Firsts e Follows. Assim, construimos uma tabela com os Firsts:

First(PROGRAM)	=	int | float | string | ident | print | read | return | if | for | { | break | ; | def | ε
First(FUNCLIST)	=	def
First(FUNCLIST')	=	def | ε
First(FUNCDEF)	=	def
First(PARAMLIST)	=	int | float | string | ε
First(PARAMLIST')	=	, | ε
First(STATEMENT)	=	int | float | string | ident | print | read | return | if | for | { | break | ;
First(VARDECL)	=	int | float | string
First(VARDECL')	=	[| ε
First(ATRIBSTAT)	=	ident
First(ATRIBSTAT')	=	+ | - | int_constant | float_constant | string_constant | null | ident | (| new
First(FUNCCALL)	=	ident
First(PARAMLISTCALL)	=	ident | ε
First(PARAMLISTCALL')	=	, | ε
First(PRINTSTAT)	=	print
First(READSTAT)	=	read
First(RETURNSTAT)	=	return
First(IFSTAT)	=	if
First(IFSTAT')	=	else | ε
First(FORSTAT)	=	for
First(STATELIST)	=	int | float | string | ident | print | read | return | if | for | { | break | ;
First(STATELIST')	=	int | float | string | ident | print | read | return | if | for | { | break | ; | ε
First(ALLOCEXPRESSION)	=	new
First(ALLOCEXPRESSION')	=	int | float | string
First(ALLOCEXPRESSION'')	=	[

First(ALLOCEXPRESSION''')	=	[| ε
First(EXPRESSION)	=	+ | - | int_constant | float_constant | string_constant | null | ident | (
First(EXPRESSION')	=	< | > | = | !
First(EXPRESSION'')	=	= | + | - | int_constant | float_constant | string_constant | null | ident | (
First(NUMEXPRESSION)	=	+ | - | int_constant | float_constant | string_constant | null | ident | (
First(NUMEXPRESSION')	=	+ | - | ε
First(TERM)	=	+ | - | int_constant | float_constant | string_constant | null | ident | (
First(TERM')	=	* | / | % | ε
First(UNARYEXPR)	=	+ | - | int_constant | float_constant | string_constant | null | ident | (
First(FACTOR)	=	int_constant | float_constant | string_constant | null | ident | (
First(LVALUE)	=	ident
First(LVALUE')	=	[| ε

E uma tabela com os Follows:

Follow(PROGRAM)	=	int | float | string | ident | print | read | return | if | for | { | break | ; | def | ε
Follow(FUNCLIST)	=	def
Follow(FUNCLIST')	=	def | ε
Follow(FUNCDEF)	=	def
Follow(PARAMLIST)	=	int | float | string | ε
Follow(PARAMLIST')	=	, | ε
Follow(STATEMENT)	=	int | float | string | ident | print | read | return | if | for | { | break | ;
Follow(VARDECL)	=	int | float | string
Follow(VARDECL')	=	[| ε
Follow(ATRIBSTAT)	=	ident
Follow(ATRIBSTAT')	=	+ | - | int_constant | float_constant | string_constant | null | ident | (| new
Follow(FUNCCALL)	=	ident
Follow(PARAMLISTCALL)	=	ident | ε
Follow(PARAMLISTCALL')	=	, | ε

Follow(PRINTSTAT)	=	print
Follow(READSTAT)	=	read
Follow(RETURNSTAT)	=	return
Follow(IFSTAT)	=	if
Follow(IFSTAT')	=	else | ε
Follow(FORSTAT)	=	for
Follow(STATELIST)	=	int | float | string | ident | print | read | return | if | for | { | break | ;
Follow(STATELIST')	=	int | float | string | ident | print | read | return | if | for | { | break | ; | ε
Follow(ALLOCEXPRESSION)	=	new
Follow(ALLOCEXPRESSION')	=	int | float | string
Follow(ALLOCEXPRESSION'')	=	[
Follow(ALLOCEXPRESSION''')	=	[| ε
Follow(EXPRESSION)	=	+ | - | int_constant | float_constant | string_constant | null | ident | (
Follow(EXPRESSION')	=	< | > | = | !
Follow(EXPRESSION'')	=	= | + | - | int_constant | float_constant | string_constant | null | ident | (
Follow(NUMEXPRESSION)	=	+ | - | int_constant | float_constant | string_constant | null | ident | (
Follow(NUMEXPRESSION')	=	+ | - | ε
Follow(TERM)	=	+ | - | int_constant | float_constant | string_constant | null | ident | (
Follow(TERM')	=	* | / | % | ε
Follow(UNARYEXPR)	=	+ | - | int_constant | float_constant | string_constant | null | ident | (
Follow(FACTOR)	=	int_constant | float_constant | string_constant | null | ident | (
Follow(LVALUE)	=	ident
Follow(LVALUE')	=	[| ε

Para uma gramática ser LL(1), suas produções do tipo `A → α | β` devem seguir as seguintes regras:

1. `First(α) ∩ First(β) = ∅`
2. Se `β` produz `ε` em 1 ou mais passos de derivação, então `First(α) ∩ Follow(A) = ∅`

3. Se `α` produz `ε` em 1 ou mais passos de derivação, então `First(β) ∩ Follow(A) = ∅`

Como nossa gramática agora está fatorada à esquerda, sabemos que a regra `1` já está contemplada, pois o objetivo da fatoração é justamente impedir que um símbolo não terminal não gere duas produções que tenham o mesmo prefixo.

Similarmente, as regras `2` e `3` estão contempladas pois nossa gramática é livre de recursão à esquerda. Isso ocorre pois, se `First(α) ∩ Follow(A) ≠ ∅` for verdade, precisamos de uma produção no estilo de `A → Aα`. Além disso, nossa gramática não possui produções do tipo `A → α | β` onde α ou β produzam `ε` em 1 ou mais passos de derivação.

Implemenação do Analisador Sintático

Como a linguagem utilizada é puramente funcional, a implementação do analisador sintático difere dos algoritmos apresentado em aula, de forma que possamos utilizar melhor as vantagens da programação funcional, ao invés de tentar recriar as funcionalidades de linguagens imperiais.

A análise sintática é realizada juntamente com a análise léxica (uma possibilidade mencionada em aula), e organiza o código fonte em uma árvore (abordagem Top-Down). Ao fazer as análises léxica e sintática ao "mesmo tempo", utilizamos menos recursos e menos tempo, além de tornar o código do compilador bastante legível.

A representação de cada nodo da àrvore pode ser encontrada nos arquivos `src/Syntax/Expression.elm` e `src/Syntax/Statement.elm`, enquanto a raíz da àrvore está em `src/Syntax/Program.elm`. Vemos que a árvore começa como um único `Statement` ou com uma lista de funções. A árvore então segue com os nodos de statements e expressions, como descrito no primeiro relatório.

Como não usamos o algoritmo da tabela de derivação apresentado em aula, não é possível informar em qual linha e coluna da tabela estamos. Porém, ainda conseguimos indicar quais símbolos estávamos esperando, e qual produção estávamos tentando. Para tal, foi usado o módulo avançado da biblioteca de *parsing* descrita no primeiro relatório, que nos permite maiores detalhes nas descrições dos erros. Esta melhoria resultou no arquivo `src/CCParser.elm`, que é responsável por representar os contextos e problemas existentes, além de transformar os erros em algo apresentável ao usuário do compilador. Os erros se dão em algo do tipo *"Achei um erro! Aqui é o que eu estava esperando ver, e eu estava tentando ler esta estrutura"*. Por exemplo, se recebemos algo como `def printMenu( {}`, o analisador nos mostra um erro como *"Estava esperando ver um parênteses fechando (`)`), e eu estava no meio de uma declaração de função (1:1)"*, onde *(1:1)* é a linha e coluna onde a declaração de função começou.

Foi inserido um novo exemplo de uso da linguagem no arquivo `examples/mathOperations`.