PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ
ESCOLA POLITÉCNICA

DIEGO BARRETO PEDROSO SIMÕES
MURIEL TRAMONTIN VON LINSINGEN
VINICIUS RAMOS GARCIA
WILLIAM HOEFLICH WOINAROWSKI

PROJETO DE COMPILADOR

CURITIBA
2023

DIEGO BARRETO PEDROSO SIMÕES
MURIEL TRAMONTIN VON LINSINGEN
VINICIUS RAMOS GARCIA
WILLIAM HOEFLICH WOINAROWSKI

PROJETO DE COMPILADOR

Projeto apresentado à disciplina de Linguagens Formais e Compiladores da Graduação em Engenharia da Computação da Pontifícia Universidade Católica do Paraná, como requisito parcial de avaliação.

Orientador: Prof. Frank Coelho de Alcantara

CURITIBA
2023

SUMÁRIO

1 PROJETO DE COMPILADOR

1.1 CONTEXTO

Este projeto tem como objetivo o desenvolvimento de um compilador para uma nova linguagem de programação a ser estruturada, e seu foco será nas plataformas de sistemas embarcados baseados no microcontrolador ATmega2560. Para o desenvolvimento do compilador serão utilizados alguns recursos de *software*, como a plataforma de desenvolvimento online Replit, o Visual Studio Code e o Hercules para a gravação do Assembly gerado no microcontrolador, e de hardware, que será composto pela placa Arduino Mega, onde todos os exemplos gerados pela linguagem serão executados.

1.2 DEFINIÇÃO DA LINGUAGEM

A linguagem a ser utilizada neste compilador está baseada na gramática fornecida a seguir, onde um programa consistirá em um bloco de declarações definidos pela tabela:

BLOCO/DECLARAÇÃO	**CONTÉM**
PROGRAM	BLOCK
BLOCK	{DECLS STMTS}
DECLS	DECLS DECL | ϵ
DECL	TYPE ID
TYPE	TYPE [NUM] | BASIC
STMTS	STMTS STMT | ϵ

O lexema **BASIC** expressa os tipos básicos da linguagem:

STMT	IF (BOOL) STMT
	IF (BOOL) STMT ELSE STMT
	WHILE (BOOL) STMT
	DO STMT WHILE (BOOL)
	BREAK

	BLOCK

Cada bloco ou declaração contém alguns parâmetros, que também podem ser blocos ou declarações, e representam o acervo de palavras restritas aceito pela linguagem.

As produções para as expressões tratam da associatividade e precedência de operadores. Elas usam um não-terminal para cada nível de precedência e um não-terminal, **FACTOR**, para as expressões entre parênteses, identificadores, referências de arranjos e constantes:

BOOL	BOOL || JOIN | JOIN
JOIN	JOIN && EQUALITY | EQUALITY
EQUALITY	EQUALITY == REL | EQUALITY != REL | REL
REL	EXPR < EXPR | EXPR <= EXPR | EXPR >= EXPR | EXPR > EXPR | EXPR
EXPR	EXPR + TERM | EXPR – TERM | TERM
TERM	TERM * UNARY | TERM / UNARY | UNARY
UNARY	! UNARY | - UNARY | FACTOR
FACTOR	(BOOL) | NUM | REAL | TRUE | FALSE

O lexema **NUM** indica números inteiros, o lexema **REAL** indica números de ponto flutuante com 16 bits segundo o padrão IEEE-754, conhecido como meia precisão. Todas as operações matemáticas, serão realizadas com a precisão definida no padrão IEEE – 754 para 16bits.

Para a interação com o hardware foram propostas as seguintes regras a serem associadas ao lexema **BASIC**:

STMT	TYPE ID = NUM | REAL | TRUE | FALSE
	DIGITAL_READ (PIN)
	DIGITAL_WRITE (PIN, NUM)
	ANALOG_READ (PIN)
	ANALOG_WRITE (PIN, NUM)
	PIN_MODE (NUM, PIN_TYPE)
	DELAY (NUM)

O lexema **NUM** indica números inteiros, e o lexema **PIN_TYPE** indica a configuração de leitura ou escrita do pino do microcontrolador:

PIN_TYPE	OUTPUT | INPUT

A partir desta gramática podemos criar programas com interação direta com o *hardware* para a execução de diversas atividades, porém, com algumas restrições a serem abordadas no próximo tópico.

1.3 RESTRIÇÕES DA LINGUAGEM

Está linguagem não irá compreender estruturas mais complexas como:

- Estrutura de repetição “FOR”
- Estrutura condicional composta “IF stmt ELSEIF stmt ELSE stmt”
- Estrutura condicional composta “SWITCH stmt CASE expr”
- Recursividade
- Funções

1.4 EXEMPLOS DE APLICAÇÃO

A seguir estão disponíveis 3 exemplos de aplicação da linguagem de programação contendo interação com o hardware do microcontrolador ATMega2560:

1. Piscar um led (pisca o led em intervalos de 1 segundo):

```
NUM PIN_10 = 10

PIN_MODE (PINO_10, OUTPUT)

WHILE (TRUE) {
        DIGITAL_WRITE (PINO_10, TRUE)
        DELAY (1000)
        DIGITAL_WRITE (PINO_10, FALSE)
        DELAY (1000)
}
```

2. Liga led com botão (acende o led somente com o botão pressionado):

```
NUM PIN_9 = 9
NUM PIN_10 = 10

PIN_MODE (PINO_9, INPUT)
PIN_MODE (PINO_10, OUTPUT)

WHILE (TRUE) {
        IF (DIGITAL_READ (PINO_9)) {
                DIGITAL_WRITE (PINO_10, TRUE)
        }
}
```

3. Indicador de intensidade de um potenciômetro (aciona os leds conforme ocorre alteração no potenciômetro):

```
NUM PIN_9 = 9
NUM PIN_10 = 10
NUM PIN_11 = 11

PIN_MODE (PINO_9, INPUT)
PIN_MODE (PINO_10, OUTPUT)

WHILE (TRUE) {
        IF (ANALOG_READ (PIN_9) < 256){
                DIGITAL_WRITE (PINO_10, FALSE)
                DIGITAL_WRITE (PINO_11, FALSE)
        }
        IF (ANALOG_READ (PIN_9) >= 256){
                DIGITAL_WRITE (PINO_10, TRUE)
        }
        IF (ANALOG_READ (PIN_9) >= 512){
                DIGITAL_WRITE (PINO_10, FALSE)
                DIGITAL_WRITE (PINO_11, TRUE)
        }
}
```

1.5 PROJETO DO COMPILADOR

Todo o projeto do compilador será realizado em Python através da IDE Replit e do Visual Studio Code, passando pela análise léxica, sintática e semântica. Para a gravação do Assembly gerado será utilizado o *software* Hercules.

O projeto do compilador, desde sua definição, documentação e códigos está disponível em um repositório público hospedado no GitHub:

https://github.com/thewillboy/compilador

1.6 DESENVOLVIMENTO DO CÓDIGO

Devido a complexidade de uma estrutura de código para a análise léxica, sintática e semântica de uma linguagem de programação, optou-se pela pesquisa de alternativas em Python para facilitar o processo de criação desta nova linguagem, ou pelo menos amenizar alguns problemas de desenvolvimento que poderiam levar a erros.

A alternativa escolhida para esse processo foi a biblioteca SLY (Sly Lex Yacc) do Python, que é focada no desenvolvimento de parsers e compiladores através da definição de tokens e o uso do algoritmo LALR para sua validação.

Com o auxílio da biblioteca SLY, podemos focar no desenvolvimento dos tokens necessários para a linguagem, criando-os de forma a serem validados com o mínimo de ambiguidades possíveis e criando uma árvore sintática que facilite a compreensão de como eles serão interpretados.

1.7 ANALISADOR LÉXICO

Os aspectos léxicos do programa são tratados nesta fase. Aqui entra em cena o analisador léxico, também conhecido como *scanner*. Ele lê o texto do programa e o divide em *tokens*, que são os símbolos básicos de uma linguagem de programação, representando palavras reservadas, identificadores, literais numéricos e de texto, operadores e pontuações.

Vamos analisar um exemplo de token chamado “BIGGER_OR_EQUAL”, que representa uma condicional de validação em que um número ou expressão numérica podem ser maiores ou iguais a outra expressão. Na linguagem criada este token foi definido utilizando uma expressão regular:

BIGGER_OR_EQUAL = r'>='

Essa expressão regular irá ser utilizada pelo analisador léxico para varrer o código e encontrar sua aplicação, o que refletirá na criação de um token usado posteriormente na análise sintática e na geração do código de máquina.

1.8 ANALISADOR SINTÁTICO

Etapa realizada pelo analisador sintático, ou parser. Este analisador trabalha lendo os *tokens* gerados pelo analisador léxico e os agrupando de acordo com a estrutura sintática da linguagem. O resultado é chamado de árvore de derivação.

Utilizando o exemplo anteriormente citado do *token* “BIGGER_OR_EQUAL”, este não executa nenhuma ação sem que esteja em um conjunto de outros tokens, ou seja, um agrupamento deles que respeita uma estrutura sintática definida na linguagem. A seguir temos um exemplo de como ele deve ser utilizado na linguagem desenvolvida neste documento:

expr BIGGER_OR_EQUAL expr

O *token* está entre duas expressões, formando assim uma estrutura condicional que pode ser utilizada dentro de um IF ou WHILE, retornando um booleano referente ao seu resultado.

1.9 TOKENS E ÁRVORE SINTÁTICA

Através da execução do analisador léxico em um trecho de código da linguagem desenvolvida temos como retorno a criação de um token similar ao exemplo a seguir:

TYPE_INT a = 1

Token(type='TYPE_INT', value='TYPE_INT', lineno=1, index=0, end=8)

Token(type='NAME', value='a', lineno=1, index=9, end=10)

Token(type='=', value='=', lineno=1, index=11, end=12)

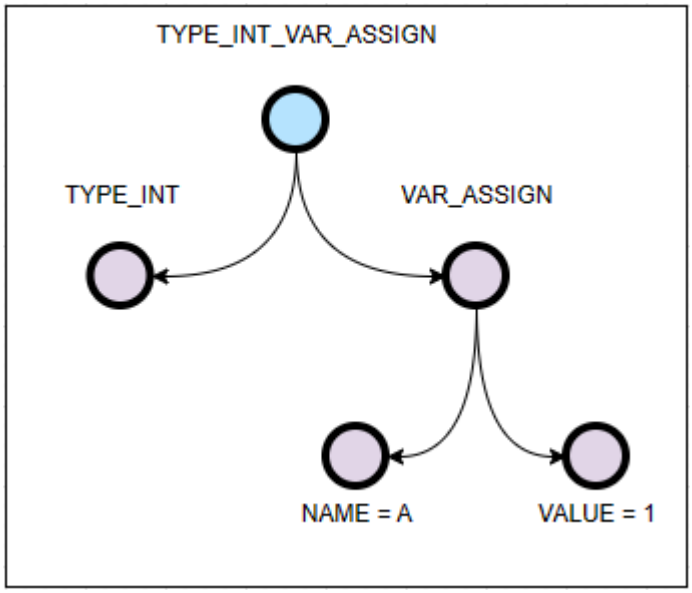
Token(type='INT', value=1, lineno=1, index=13, end=14)

Estes *tokens* contém as informações do seu tipo, valor, linha, posição inicial e final em um *array*.

Como resultado da análise semântica do mesmo trecho de código temos a seguinte árvore sintática, que pode ser utilizada para validarmos como o código será executado:

('type_int_var_assign', 'TYPE_INT', ('var_assign', 'a', 1))

Aplicando uma representação visual nessa árvore vamos ter o seguinte resultado:



1.10 ANALISADOR SEMÂNTICO

Aspectos semânticos são tratados aqui. Estes aspectos levam em consideração o sentido do programa. Sua implementação exige a inserção de diversas validações no código, que irão testar o sentido do que se deseja realizar, porém, dada sua complexidade este item não será abordado neste compilador, pois a análise sintática já efetua um filtro muito simples que irá possibilitar a futura geração do código de máquina.

1.11 TESTES DO CÓDIGO

O código do compilador está disponível no repositório público ou da plataforma Replit:

- GitHub: https://github.com/thewillboy/compilador
- Replit: https://replit.com/@WilliamHoeflich/Compilador

A execução do programa está dividida em dois modos, um para a análise de arquivos de texto contendo o código criado ou uma console para inserirmos o trecho de código a ser analisado.

O resultado da análise de um arquivo será outro arquivo contendo os tokens e as árvores sintáticas geradas, porém, somente será criado caso não existam erros presentes na estrutura do código.

1.12 FALHAS E AMBIGUIDADES DO CÓDIGO MAPEADAS

Algumas falhas e possibilidades de ambiguidades foram identificadas durante a análise sintática dos tokens criados, sendo necessário ajustar o código da linguagem para evitá-las:

- Para os operadores de condição e repetição foi necessário adicionar um token “END” para delimitar o final das suas declarações, entretanto, mesmo dessa forma algumas declarações ainda não estão gerando a árvore sintática correta:

  IF (TRUE) THEN { DIGITAL_READ(a) } END

As falhas identificadas foram separadas em um arquivo de teste presente no repositório público desta linguagem.

REFERÊNCIAS

ATMega2560. Disponível em: <https://www.microchip.com/en-us/product/ATMEGA2560>. Acesso em: 16 abr. 2023.

Análise léxica, sintática e semântica. Disponível em: <https://autociencia.blogspot.com/2019/03/automatos-lexico-sintaxe-semantica.html>. Acesso em: 16 abr. 2023.

IEEE-754. Disponível em: <https://www.mathworks.com/help/coder/ug/what-is-half-precision.html>. Acesso em: 16 abr. 2023.

SLY (Sly Lex Yacc). Disponível em: <https://sly.readthedocs.io/en/latest/sly.html>. Acesso em: 14 mai. 2023.

Let's Make a programming language! 1. the Lexer. Disponível em: < https://replit.com/talk/learn/Lets-Make-a-programming-language-1-the-Lexer/87022>. Acesso em: 14 mai. 2023.