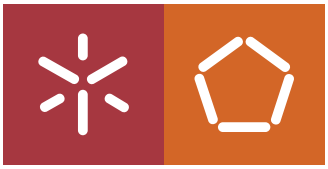


Universidade do Minho
Escola de Engenharia

Processamento de Linguagens – Trabalho Prático

Compilador de Pascal

Grupo 4

Humberto Gil Azevedo Sampaio Gomes	A104348
José António Fernandes Alves Lopes	A104541
José Rodrigo Ferreira Matos	A100612

1 de junho de 2025

Resumo

Ao longo do último semestre, foi desenvolvido, no âmbito da UC de Processamento de Linguagens, um compilador de Pascal Standard [1] para a *stack machine* EWVM [2]. O desenvolvimento dos analisadores léxico e sintático foi feito em duas fases: em primeiro lugar, criaram-se versões mínimas destes componentes, funcionais para um conjunto reduzido de programas de exemplo, que depois foram enriquecidos com informação do *standard* ISO, garantido que o compilador desenvolvido implementava a linguagem com correção. Já a análise semântica foi feita diretamente com informação da ISO, bem como a geração de código máquina foi feita diretamente com base na documentação da EWVM. Adicionalmente, implementaram-se mensagens de erro descritivas, diversos testes do compilador, e otimização do código, tanto do grafo de sintaxe abstrata como do código EWVM gerado. Os resultados obtidos, apesar de irem além do que foi pedido pelo enunciado do trabalho prático proposto, não cumprem por completo o *standard* ISO.

1 Arquitetura e Utilização do Compilador

Abaixo, apresenta-se um esquema da arquitetura do compilador desenvolvido:

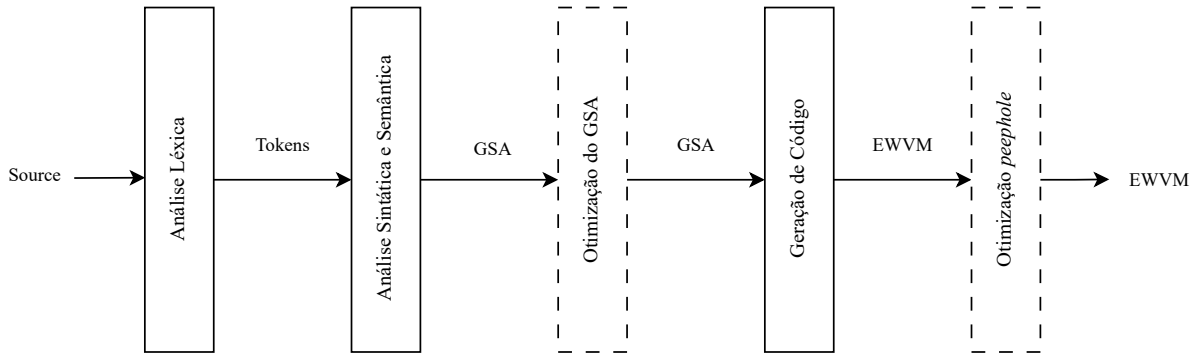


Figura 1: Arquitetura do compilador desenvolvido.

A compilação começa com a divisão do código fonte em lexemas, que depois são utilizados para a análise sintática. Por diversos motivos explicados posteriormente, a análise semântica é feita com ações semânticas durante a análise sintática, e origina um grafo de semântica abstrata (GSA). Opcionalmente, antes da geração de código EWVM, o GSA pode ser otimizado. Depois, segue-se a geração de código EWVM, que pode ainda ser otimizado por métodos de otimização *peephole*.

A orquestração destes vários componentes é feita pelo ponto de entrada do compilador, que analisa os argumentos da linha de comandos para determinar o ficheiro de fonte, se o código

deve ou não ser otimizado, se devem ou não ser exportados símbolos de *debug*, e para que ficheiro o código EWVM gerado deve ser exportado. Por omissão, o compilador lê o código fonte do `stdin`, e escreve o código gerado para o `stdout`. Segue a lista completa de opções de linha de comandos:

```
usage: python -m plpc [-h] [-o O] [-O] [-g] [file]

Compile Pascal for the EWVM.

positional arguments:
  file        path to the file to compile

options:
  -h, --help  show this help message and exit
  -o O        output assembly file
  -O          optimize generated code
  -g          add debug symbols
```

2 Análise Léxica

A análise léxica é a primeira fase de compilação, onde o código fonte é divido em lexemas. Para implementar o analisador léxico, o `ply.lex` foi utilizado [3]. Com esta ferramenta, basta definir o formato de cada tipo de lexema com uma expressão regular, e definir a prioridade do reconhecimento destes lexemas.

Definir as expressões regulares de cada tipo de lexema foi trivial, mas alguns cuidados foram necessários. Em primeiro lugar, a *flag* `re.IGNORECASE` foi utilizada no analisador léxico. Como Pascal é uma linguagem *case-insensitive*, esta *flag* permite que não seja necessário, em todas as expressões regulares, utilizar construções RegEx como `(?i)`. Ademais, todas as palavras reservadas e constantes numéricas são terminadas por uma *word boundary* (ex: `PROGRAM\b`), para exigir a existência de caracteres não alfanuméricos a separar estes *tokens*. Deste modo, *inputs* como `FORFOR` são reconhecidos com um identificador, em vez de duas palavras reservadas `FOR`.

Depois, procedeu-se à definição das prioridades de reconhecimento dos vários lexemas. Para tal, muitos dos tipos de lexema foram definidos na forma de função, para a sua ordem de reconhecimento ser a ordem de declaração das funções. Segue-se, abaixo, um exemplo de uma destas declarações:

```
def t_INTEGER(self, t: ply.lex.LexToken) -> ply.lex.LexToken:
```

```
    r'[0-9]+\b'
    t.value = int(t.value)
    return t
```

Foi necessário garantir que lexemas cujo valor poderia constituir um prefixo de um lexema de outro tipo eram declarados depois desse outro tipo de lexema. Logo, todas as palavras reservadas foram declaradas antes da declaração de um identificador, e a definição de uma constante real precede a de uma constante inteira. Foi possível, devido à inexistência de conflitos, definir alguns lexemas utilizando constantes em vez de funções (ex: `t_RANGE = r'\.\.'`). Note-se que estes lexemas, incluindo o exemplo `RANGE`, têm prioridade de reconhecimento sobre os literais [3]. Logo, não haverá conflitos entre os lexemas `'..'` e `'.'`.

Por último, é necessário manipular alguns lexemas após a sua deteção. Por exemplo, constantes reais e inteiras devem ser convertidas para `float`s e para `int`s, respetivamente. Ademais, é necessário processar constantes textuais para remoção das plicas, e para transformação de sequências de duas plicas no seu conteúdo em apenas uma (é assim que plicas são escapadas em Pascal). Estas transformações são feitas em funções como a apresentada acima.

Um detalhe de implementação a mencionar é que o analisador léxico foi implementado como uma classe Python, sendo este o motivo para a função apresentada acima ter `self` como parâmetro. Deste modo, é possível criar vários analisadores léxicos num único processo, sem qualquer estado global, o que é essencial para a implementação de testes unitários, nos quais se entrará em mais detalhe posteriormente neste documento.

Para o desenvolvimento do analisador léxico, a leitura do *standard* ISO que define Pascal ajudou apenas a polir algumas arestas. A título de exemplo, permitiu corrigir a definição de comentário, visto que, em Pascal, um comentário começado por `(*` pode terminar em }, ou um comentário começado por { pode acabar em `*)`. Ademais, o *standard* ensinou o grupo de trabalho sobre caracteres alternativos, [1] que foram implementados, bem como alguns tipos de *statement* menos conhecidos (como `WITH`), que utilizavam lexemas que também foram definidos. Por último, a leitura do *standard* também fez com que `NIL` fosse considerado um lexema, em vez de um identificador pré-definido na tabela de símbolos (como, por exemplo, `maxint`).

No anexo 12.1, pode ser consultada a definição completa do analisador léxico.

[1] Devido aos *character-sets* limitados de alguns sistemas contemporâneos ao desenvolvimento de Pascal, a linguagem permite substituir sequências de alguns caráteres por outros. Por exemplo: `'(.'` → `'['`.

3 Análise Sintática

Para a construção do analisador sintático, o `ply.yacc` [3] foi utilizado. A construção inicial da gramática de Pascal foi relativamente simples: partiu-se do símbolo não terminal `program`, que foi sendo decomposto recursivamente até se chegarem aos símbolos terminais. Durante este processo, foram identificadas estruturas iguais utilizadas em diferentes regras, que foram agrupadas no mesmo símbolo não terminal. A título de exemplo:

- Um programa pode ser decomposto num cabeçalho de programa e num bloco;
- Um bloco pode ser decomposto numa secção de declaração de constantes, numa secção de declaração de tipos, numa secção de declaração de variáveis, ...;
- Uma secção de declaração de constantes pode ser vazia, ou decomposta na palavra reservada `CONST` e numa lista de declarações uma ou mais de constantes;
- ...

Com este método *top-down*, a construção da gramática foi simples. A única parte de complexidade superior foi a definição da gramática das expressões. Como os operadores unários `+` e `-` apenas podem ocorrer no início de uma expressão, foi necessária a distinção entre os símbolos não terminais para os primeiros termos e fatores, diferentes dos símbolos não terminais para os termos e fatores seguintes.

A gramática inicialmente construída, apesar de ser simples, precisou de ser consideravelmente modificada para cumprir o *standard* ISO. Muitos dos problemas corrigidos prendiam-se com a possibilidade / impossibilidade de, dependendo do contexto, ser possível ter *trailing commas / semicolons*. Outra grande mudança feita foi a alteração da prioridade dos operadores binários nas expressões. Inicialmente, como acontece em outras linguagens, foi dada prioridade máxima aos operadores lógicos, que eram seguidos dos relacionais, que eram seguidos dos aritméticos para multiplicação e divisão, que, por fim, eram seguidos dos aritméticos para soma e subtração. No entanto, em Pascal, o operador `AND` deve ter tanta prioridade quanta a do `*`, e o `OR` tanta prioridade quanta a do `+`, o que exigiu alterações à gramática.

Para melhorar o desempenho do compilador, seguindo as recomendações da documentação do `ply.yacc` [3], as várias produções dos símbolos não terminais foram frequentemente colocadas em diferentes funções, para evitar `if`-*statements* que procuram detetar a produção utilizada. Segue-se um exemplo desta técnica, a definição de uma lista de expressões separada por vírgulas:

```
def p_expression_list_single(self, p: ply.yacc.YaccProduction) -> None:
```

```
    '''
    expression-list : expression
    '''
    p[0] = [p[1]]

def p_expression_list_multiple(self, p: ply.yacc.YaccProduction) -> None:
    '''
    expression-list : expression-list ',' expression
    '''
    p[1].append(p[3])
    p[0] = p[1]
```

No exemplo acima, observa-se também que, na gramática, foi utilizada recursividade à esquerda. Deste modo, a *stack* do autómato LALR não precisa de aumentar significativamente para o processamento de estruturas recursivas, resultando num melhor desempenho e permitindo o processamento de estruturas recursivas de grande profundidade.

O exemplo da definição de uma lista apresentado acima também revelou algumas limitações do `ply.yacc` encontradas na realização deste trabalho. O `ply.yacc` gera um *parser* com base numa gramática na forma Backus-Naur (BNF) [3]. No entanto, teria sido mais simples definir a gramática se fosse possível fazê-lo na forma de Backus-Naur estendida, onde seria possível, sem a verbosidade acima, definir conceitos mais complexos como a ocorrência opcional de um símbolo, listas de ocorrências de um símbolo, *etc.*. Assim, também seria possível remover muito código duplicado do *parser* como, por exemplo, as várias declarações de listas de diferentes símbolos presentes na gramática.

A gramática utilizada pode ser consultada no anexo 12.2.

4 Análise Semântica

A análise semântica, no compilador desenvolvido, é feita em ações semânticas, ou seja, código associado a produções na gramática, que é executado quando uma redução é feita por uma dessas produções. Deste modo, as análises sintática e semântica são feitas em simultâneo, o que traz várias vantagens. Em primeiro lugar, não é necessário implementar várias estruturas intermédias: em vez de uma árvore de sintaxe abstrata (AST), à qual é depois adicionada informação semântica, uma única estrutura com informação semântica basta. Ademais, ao fazer estes dois processos em simultâneo, evita-se a iteração por mais uma estrutura intermédia, possivelmente melhorando o desempenho do compilador. Por último, para informar o utilizador da localização dos erros semânticos no código, seria necessário armazenar a localização de todos os lexemas na árvore de sintaxe abstrata, ou seja torná-la uma árvore de sintaxe concreta, o que

aumentaria a sua complexidade. Logo, decidiu-se que a análise semântica seria feita em ações semânticas, e que o resultado do *parser* seria um grafo de semântica abstrata, ou seja, uma AST enriquecida com referências semânticas (referência a declarações de variáveis, propagação de tipos, *etc.*).

Gerar um GSA durante o *parsing* só é possível porque, em Pascal, para referenciar um objeto (constante, variável, *etc.*), é necessário que ele já tenha sido declarado antes. Logo, tendo em conta que o *parser* utiliza recursividade à esquerda, quando um objeto é declarado, é adicionado a uma tabela de símbolos, e quando o código o referencia, é realizada uma consulta a esta tabela. Esta estrutura de dados consiste numa pilha de tabelas de *hash*, onde cada elemento da pilha corresponde a um *scope* aninhado, e associa os nomes de objetos nesse *scope* aos objetos em si. Considere-se o exemplo abaixo:

```
program SymbolTableExample;
var
    x, y: integer;

procedure exampleProcedure(z: integer);
var
    arr: array [1..10] of integer;
begin
    ;
end;

begin
    y := y + 1;
end.
```

No corpo do procedimento, o estado da tabela de símbolos será o seguinte:

`{ 'z': VariableDefinition(...), 'arr': VariableDefinition(...) }`
`{ 'x': VariableDefinition(...), 'y': VariableDefinition(...) }`

Para interrogar a tabela de símbolos, por exemplo, quando uma variável é utilizada, começa-se por pesquisar o identificador referenciado no topo da *stack*, e vai-se descendo até ele ser encontrado. Caso não seja encontrado, ocorre um erro semântico.

Na criação do GSA, estas referências aos objetos são substituídas por arestas para esses objetos. Abaixo, pode observar-se uma versão simplificada do GSA do programa acima:

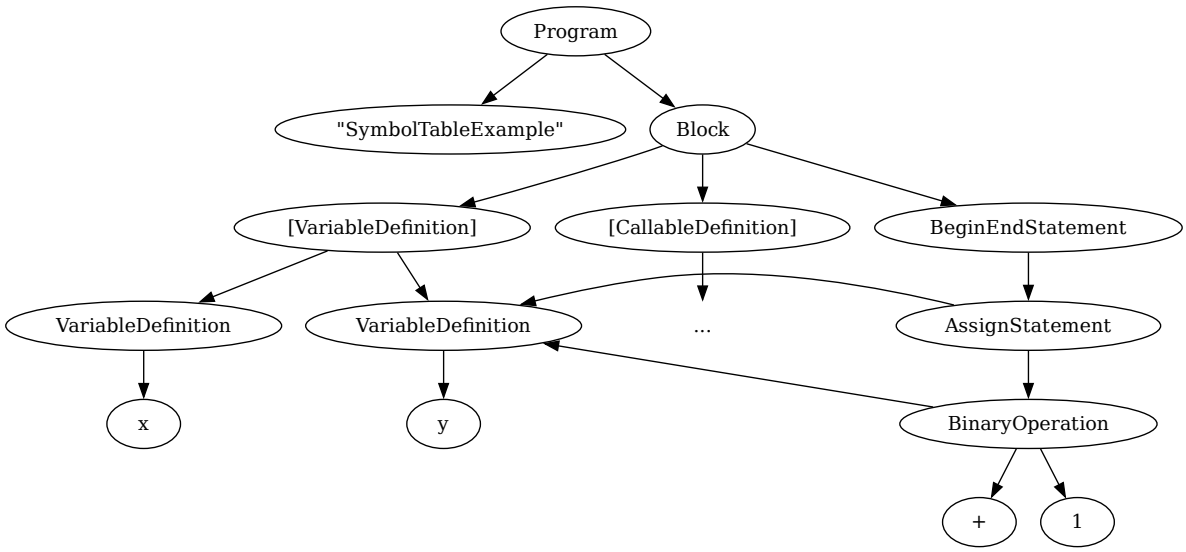


Figura 2: GSA do programa apresentado após remoção de informação de tipos.

Também é durante a análise semântica que se garante a correção de tipos do programa Pascal. Para o fazer, sempre que se encontram atribuições, indexações, ou outras operações lógicas e aritméticas, um módulo `typechecker`, com funções auxiliares para verificação de compatibilidade de tipos, é utilizado para verificar se as operações no código são possíveis dados os tipos dos operandos. Na construção do GSA, esta informação de tipos é adicionada aos vários nós, para não ter de ser recalculada durante as fases de otimização e geração de código. Abaixo, segue-se o GSA completo da expressão `1 + 5.0 < 7.0`:

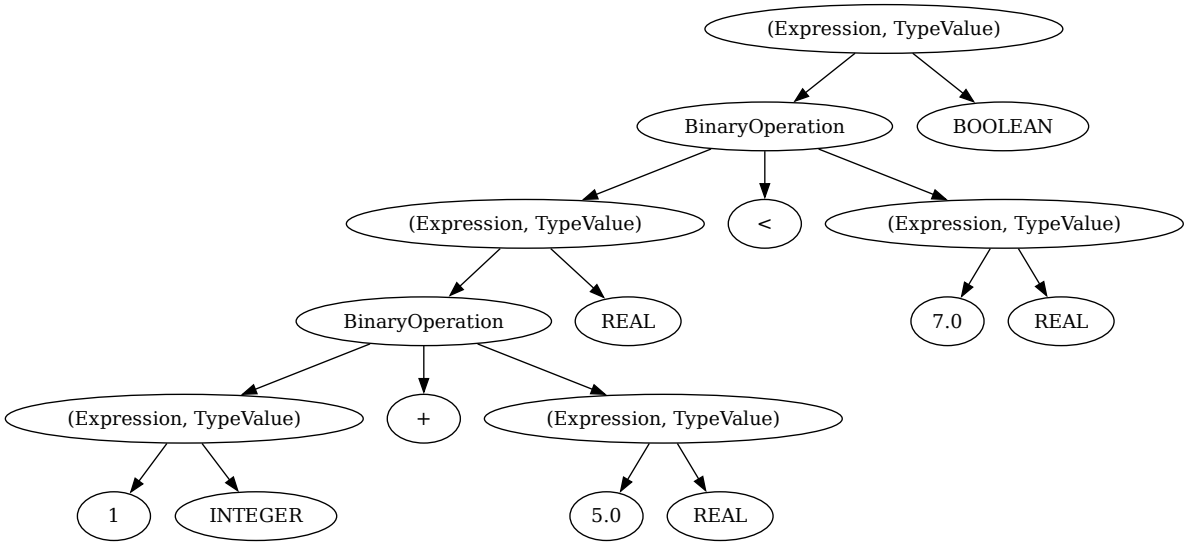


Figura 3: GSA de `1 + 5.0 < 7.0` com informação de tipos.

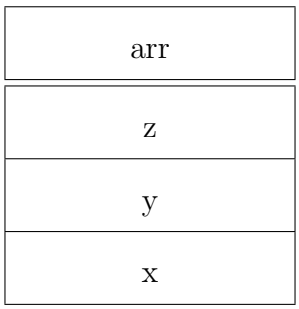
O analisador semântico foi, sem dúvida, o componente cujo desenvolvimento mais demorado foi, especialmente devido a todos os cuidados necessários com o sistema de tipos: foi necessário um grande esforço para pensar em todos os erros de tipo possíveis, e como os detetar corretamente. Ademais, devido à API do `ply.yacc`, não foi possível fazer validação estática de tipos Python com ferramentas como o `mypy` [6], pelo que foi também demorado garantir, manualmente, a correção de tipos do código Python nas produções.

5 Geração de Código

Antes de começar a geração de código, foi necessária a criação de algumas ferramentas para ser possível fazê-la corretamente e permitindo a implementação posterior de otimizações *peephole* (ver 6.2). Por esta razão, a geração de código não é feita diretamente para uma *string*, foi necessário construir uma representação intermédia do código EWVM, com tipos de dados para a representação de *labels*, de instruções, e de comentários. Ademais, também antes da geração de código em si, foi necessário criar um mecanismo para geração de novas *labels*: cada *scope* tem o seu gerador de *labels*, que as gera com base num contador simples. Para não haver conflitos de nome entre *labels* de diferentes *scopes*, todas as *labels* são prefixadas pelo nome do seu *scope*.

O facto da máquina alvo ser uma *stack machine* simplificou consideravelmente o processo de geração de código, visto que permitia a fácil concatenação de diferentes partes do programa, exigindo apenas que fosse assegurado que cada parte do programa deixava a *stack* num estado válido para a parte seguinte. Também não foi necessário a implementação de qualquer mecanismo de alocação de registos, simplificando assim a geração de código.

Antes de se começar a gerar qualquer código, foi necessário definir a organização das variáveis na pilha e a *calling convention* a utilizar. Foi decido que os *arrays* seriam alocados na *heap*, assim simplificando a geração de código, visto que qualquer variável ocuparia uma posição na pilha da máquina virtual. Depois, decidiu-se que as variáveis de um *scope* seriam colocadas na pilha depois do seu *frame pointer*, pela ordem em que são declaradas. O mesmo não é verdade para o valor de retorno e os argumentos de uma função ou procedimento, que são colocados, nesta ordem, antes do *frame pointer*, permitindo a sua colocação na pilha antes da execução da instrução `CALL`. Por exemplo, no procedimento `exampleProcedure` do programa previamente apresentado, o estado da pilha será o seguinte, onde o traço horizontal duplo representa o *frame pointer*:



Com esta convenção definida, a geração de código foi trivial, e implementada de um modo *top-down*:

- Para gerar o código do programa, gera-se, em primeiro lugar, o código do seu bloco principal, e só depois o código dos procedimentos e das funções.
- O bloco do programa é precedido pela instrução `START` e terminado pela instrução `STOP`, enquanto que um bloco de uma função / de um procedimento começa com a sua *label*, e termina com a instrução `RETURN`. No conteúdo do bloco, gera-se o código para as declarações das variáveis, e só depois para os *statements*.
- Declarações de variáveis são baseadas na geração de constantes: para gerar variáveis escalares e *strings*, basta colocar na *stack* o valor por omissão dessa variável. Variáveis que são *arrays* já são mais complexas, exigindo a alocação de memória na *heap* (com a instrução `ALLOC`) e a inicialização da memória alocada com um ciclo. No fim do seu *scope*, para eliminar os *buffers* alocados na *heap*, a instrução `POPST` é utilizada tantas vezes quantas há variáveis do tipo `array` no *scope*.
- Constantes são geradas com uma instrução da forma `PUSH(I|F|S)`, conforme o seu tipo.
- Diferentes tipos de *statements* são gerados de diferente modo, e a geração do código de cada *statement* pode ser consultada no anexo 12.3.
- Para gerar chamadas a funções e procedimentos, os valores das expressões dos argumentos são colocados na pilha, e a função / procedimento é depois chamado (com as instruções `PUSHA` e `CALL`). Quando o controlo de fluxo é devolvido, é necessário executar a instrução `POP N` para remover os elementos colocados na pilha pela função / pelo procedimento. O número de elementos é a soma do número de parâmetros com o número de variáveis do objeto chamado. No caso da invocação de uma função, sobrará sempre um elemento na pilha, o valor de devolução da função, que precede todos os argumentos na pilha. Procedimentos e funções *built-in* são exceção a esta regra, visto que foi necessário gerar código especial para os mesmos.
- Para gerar o valor de expressões (operações unárias e binárias), o código para calcular o valor das subexpressões é gerado (recursivamente), e depois utilizam-se diferentes ins-

truções conforme o operador da expressão (ex: `ADD` para somas de inteiros, `NOT` para negação lógica, *etc.*).

- Leituras e escritas de variáveis são realizadas com as instruções `PUSH(G|L)` e `STOREG(G|L)`, sendo os *offsets* aos apontadores de *frame* ou global determinados de acordo com a convenção de armazenamento de variáveis definida acima. Para variáveis que são *arrays*, é necessário, em primeiro lugar, carregar o apontador para o *array* na *heap*, ação à qual se pode seguir o cálculo do índice no *array*. Para isso, para cada índice, calcula-se o valor da expressão do índice, subtrai-se o valor base do *range type* do *array* (com as instruções `PUSHI` e `SUB`), e multiplica-se, com a instrução `MUL`, esse valor pelo tamanho do *elemento* do *array*. Este *offset* é adicionado ao apontador previamente calculado (com `PADD`) e, no fim da iteração por todos os índices, tem-se o apontador para posição do *array* que se deseja ler, e que é lida com a instrução `LOAD` (ou escrita com `STORE`). Para indexar *strings*, a instrução `CHARAT` é utilizada.

Com esta metodologia, a lógica por detrás da geração de código é simples, mas o código gerado tem muitos potenciais para melhoria, pelo que são empregues técnicas de otimização *peephole* para o melhorar, como se apresenta de seguida.

Por último, durante a geração de código, são adicionados comentários com o significado de cada sequência de instruções *assembly*, e a que *statement* estas pertencem, para ajudar a depuração do código. Estes comentários, semelhantes aos símbolos de *debug*, podem ser ativados ou desativados pela linha de comandos.

6 Otimização de Código

A otimização do código é feita em duas fases. Em primeiro lugar, são feitas alterações ao GSA para o simplificar, e após a geração do código máquina, é utilizada otimização *peephole* para simplificar o código gerado, sem que seja necessário adicionar complexidade à geração de código para fazer estas transformações.

6.1 Otimizações ao GSA

O foco da otimização no GSA foram as expressões, visto que a sua natureza matemática faz com que possam ser facilmente manipuladas. Otimizações mais complexas a envolver mais do que uma instrução exigiriam a construção de grafos de controlo de fluxo, que assim ainda não seriam suficientes para realizar várias otimizações comuns, visto que muitas destas exigem

reordenação de instruções. Devido ao quão limitada é a otimização baseada apenas num GSA, compreende-se o motivo pelo qual os compiladores atuais utilizam representações intermédias numa forma *single static-assignment* (SSA) para realizarem as suas otimizações.

O algoritmo de otimização de expressões é recursivo: para uma dada expressão, começam-se por otimizar as suas subexpressões, e estas são utilizadas para construir uma nova expressão, que pode sofrer alterações com base no nó da sua raiz.

A primeira otimização implementada foi *constant folding*, ou seja, o cálculo do valor das expressões desde que as suas subexpressões sejam constantes. No entanto, como não existe reordenação de expressões e de *statements*, esta otimização é um pouco limitada. Por exemplo, considerando as árvores apresentadas abaixo, apenas a expressão `2 * 2 * x` será otimizada, devido à existência de uma subexpressão totalmente constante:

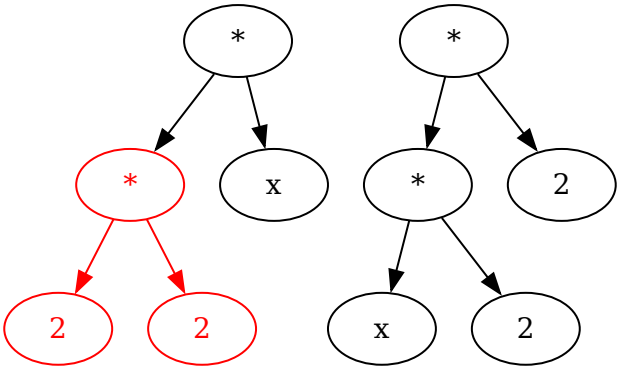


Figura 4: Árvores das expressões `2 * 2 * x` e `x * 2 * 2`, com uma árvore otimizável assinalada a vermelho.

Também foram implementadas otimizações capazes de simplificar expressões algébricas. As primeiras duas prendem-se com a negação lógica. Caso uma expressão seja constituída por negações aninhadas (`not (not x)`), será substituída pela expressão dentro das duas negações (`x`). Ademais, a negação de uma operação relacional, por exemplo, `not (x = y)`, será removida, e o operador relacional será substituído pelo seu oposto, dando origem, no exemplo acima, a `x <> y`. Por último, também foram implementadas as leis de De Morgan, pelo que expressões da forma `(not p) or (not q)` serão convertidas para `not (p and q)`, e expressões da forma `(not p) and (not q)` serão convertidas para `not (p or q)`. Muitas outras otimizações semelhantes seriam possíveis, mas não foram implementadas devido ao tempo limitado e ao esforço necessário para garantir a correção do compilador em diversos cenários.

6.2 Otimizações *peephole*

Após a geração de código, um otimizador *peephole* é utilizado para transformar sequências de instruções complexas noutras mais simples. Deste modo, não é necessário adicionar complexidade à geração de código em si para extrair um melhor desempenho da EWVM.

A primeira otimização é a substituição de sequências de instruções `PUSH(I|F) 0` por uma única instrução `PUSHN N`, onde `N` é o número de instruções substituídas. Esta otimização é particularmente útil para a inicialização das variáveis do programa, de uma função, ou de um procedimento, que são todas inicializadas a zero.

Outra otimização implementada é a transformação de multiplicações em somas. A sequência de instruções `PUSH(I|F) 2 ; (F)MUL` é substituída por `DUP; (F)ADD`. Devido ao *overhead* da execução da EWVM, esta otimização não faz qualquer diferença no desempenho do código gerado, mas trata-se de uma otimização clássica que, devido à sua relevância histórica, foi implementada. Na ausência de operações de *shift* na EWVM, não foi possível implementar outras otimizações de multiplicações e divisões por potências de base dois constantes.

Por último, também se implementou uma otimização para reduzir o número de acessos a variáveis: a sequência de instruções `STORE(L|G) N ; PUSH(L|G) N` é substituída por `DUP 1 ; STORE(L|G) N`, evitando a leitura de uma variável que se acabou de escrever para memória. Esta otimização é útil, por exemplo, na sequência de *statements* Pascal `x := ... ; y := x + ...`.

7 Erros Descritivos

Quando um compilador encontra erros no código fonte, deve apresentá-los ao utilizador. A qualidade destes erros é crucial, devendo ajudar o utilizador a facilmente identificá-los e corrigi-los. No compilador desenvolvido, procurou-se apresentar os erros léxicos, sintáticos, semânticos, e de geração de código de modo a providenciar a melhor experiência possível ao utilizador.

Durante a análise léxica, o *lexer* apresenta ao utilizador sequências de caracteres que não foram reconhecidas, ao contrário de um erro por caractere, como era feito nas aulas práticas da UC. Deste modo, caso o código fonte contenha vários caracteres inválidos em sequência, o utilizador não se deparará com dezenas de erros. Ademais, o analisador léxico é capaz de ignorar estas sequências inválidas, e prosseguir com a análise léxica. Abaixo, apresenta-se um exemplo deste erro:

```
<stdin>:3:10: error: Lexer failed to reconize the following characters
     3 |     iteração = 0;
                ^~
```

Como se pode observar, os erros são formatados para que seja possível ao utilizador ver a mensagem do erro, a sua localização no código, e o código que causou o erro. Esta formatação foi inspirada na formatação utilizada pelas versões mais recentes do GCC [4].

Em relação à análise sintática, quando um erro é detetado, o autómato LALR é inspecionado para informar o utilizador que, apesar de um lexema de um dado tipo ter sido encontrado, o autómato esperava outros tipos de lexema:

```
<stdin>:3:17: error: Unexpected token: 'begin'. Expecting: '(', ';'
     3 | procedure hello begin
                           ^~~~~
```

Também foram adicionadas regras à gramática para reconhecer erros frequentes, cujas ações semânticas apresentam mensagens do erro com sugestões ao utilizador de como pode corrigir o seu código:

```
<stdin>:5:7: error: Did you mean to use ':='?
     5 |     x = 0;
             ^
```

Infelizmente, quando um lexema inesperado é encontrado, a análise do autómato para determinar os lexemas possíveis na posição do erro também será influenciada por estas regras de erro, e a lista de lexemas esperados não estará completamente correta. No entanto, na prática, esta lista é mais do que suficiente para o utilizador rapidamente se aperceber que, por exemplo, se esqueceu de um ponto e vírgula. No entanto, esta falta de controlo sobre o autómato de *parsing* ajudou à compreensão do motivo pelo qual vários compiladores utilizam *parsers* recursivos descendentes escritos à mão, em vez de geradores de *parsers*. [5]

Em relação à recuperação de erros, esta é muito limitada. Visto que a análise semântica é feita em simultâneo com a análise sintática, um erro de sintaxe que, por exemplo, faça com que uma constante, variável, ..., não seja declarada, fará surgir erros em cascata em todo o código que use essa constante / variável / .... Para os evitar, a recuperação de erros apenas é possível com erros mínimos, e mesmo assim, erros em cascata podem surgir.

Em relação a erros semânticos, estes são muito fáceis de apresentar. Como a análise semântica é feita em ações semânticas, é trivial, com recurso à função `lexspan`, saber a localização de cada erro. No entanto, mesmo com o *tracking* ativado, esta função não aparenta estar a

ter o comportamento correto, visto que devolve comprimentos incorretos para símbolos não terminais, o que pode ser um possível *bug* do `ply`. Por este motivo, as linhas vermelhas a indicar a localização de um erro podem estar incorretas em algumas mensagens de erro específicas. Por exemplo, na mensagem de erro abaixo, devido a esta limitação, o erro foi colocado na palavra reservada `IF`, e não na expressão errada:

```
<stdin>:4:5: error: Expression in if-statement is not boolean
     4 |     if 123 + 456 then
           ^~
```

A um nível semântico, também foram implementados diversos avisos, que não impedem a compilação, mas que avisam o utilizador que pode ter cometido um erro indesejado. Segue-se o exemplo de um destes avisos:

```
<stdin>:9:19: warning: Shadowing object with name 'bin'
     9 | function BinToInt(bin: string): integer;
                           ^~~
```

Uma possível área para melhoria do compilador seria a implementação de mais destes avisos, capazes de detetar possíveis erros no código de uma forma semelhante a um *linter*.

Durante a geração de código EWVM, alguns avisos também podem ocorrer. Estes devem-se a limitações da EWVM, que não permite a representação da totalidade do *standard* Pascal. Segue-se o exemplo de um destes avisos:

```
warning: Double quotes in string 'Hello, "quoted world"' will be removed in
         EWVM output
```

Como se pode observar, durante a geração de código, já não há informação da posição de cada estrutura no código, pelo que não é possível apresentar a localização exata deste aviso.

8 Testes Realizados

Para testar o compilador, começou-se por escrever testes unitários para o analisador léxico. No entanto, usar esta metodologia para testar as análises sintática e semântica provou-se invável. Como estas são feitas em simultâneo, todos os testes precisam de ser de programas completos. Por exemplo, caso se deseje testar se o compilador está a interpretar corretamente a expressão `x + 4`, é necessário que a variável `x` se encontre declarada, o que exige o teste de um programa completo. Consequentemente, estes apresentam GSAs de grande dimensão, não sendo viável

escrever testes unitários para aferir sobre o seu valor. No entanto, testes unitários foram utilizados para testar duas componentes utilizadas durante a análise semântica: a tabela de símbolos e o verificador de tipos. Mais tarde, foram utilizados testes unitários para testar a otimização de código.

Os restantes testes realizados foram programas Pascal, tanto os fornecidos pela equipa docente como outros da autoria do grupo de trabalho, que devem ser compilados e manualmente executados na máquina virtual. Uma possível melhoria a estes testes seria a sua automatização com ferramentas como Selenium, mas isso é algo consideravelmente fora do âmbito da UC de Processamento de Linguagens, pelo que se optou por alocar os recursos humanos disponíveis para desenvolver outras funcionalidades do compilador.

9 Resultados Obtidos

O compilador desenvolvido, tal como requisitado, é capaz de transformar programas Pascal *standard* em *assembly* EWVM.

De um ponto de vista funcional, o compilador suporta várias construções da linguagem, desde as consideradas obrigatórias pelo enunciado (declaração de variáveis, expressões aritméticas e comandos de controlo de fluxo), como também outras sugeridas pelo enunciado (procedimentos e funções). Ademais, foram também implementadas outras partes do *standard*, como declaração de *labels*, constantes e tipos de dados (*aliases*, tipos enumerados, *ranges* e até *arrays* multidimensionais). Ademais, foi implementado o tipo `string` e a função `length` que, apesar de não estarem incluídos no *standard* ISO, são necessários para a compilação dos programas de exemplo fornecidos pela equipa docente.

No entanto, há várias funcionalidades de Pascal que são implementadas de uma forma limitada, e outras que, devido a falta de tempo, não foram de todo implementadas. Uma das principais limitações na implementação existente do compilador é a impossibilidade da definição de procedimentos e funções recursivos: permiti-lo implicaria mudanças substanciais à implementação da tabela de símbolos, para ser possível, por exemplo, distinguir entre uma chamada recursiva de uma função e a definição do valor a devolver. Destaca-se também a impossibilidade de passar *arrays* como argumentos de funções e procedimentos. Em relação às funcionalidades por implementar, destacam-se os vários tipos de dados por suportar (apontadores, *records*, *sets* e *files*), bem como os operadores a si associados.

Por outro lado, o cumprimento total do *standard* ISO não foi a maior preocupação dos autores do compilador, pelo que também foram implementados diversos testes, mecanismos de

otimização de código, tanto do GSA como do código EWVM, e mensagens de erro explicativas.

Em relação à correção do compilador, este foi capaz de compilar corretamente os programas de teste fornecidos pela equipa docente, bem como outros testes construídos pelo grupo de trabalho, que testam funcionalidades específicas e complexas, onde podem residir falhas. A passagem destes testes não prova a correção completa do compilador, mas ajuda a aumentar a confiança que se tem no mesmo.

Quanto à estrutura do compilador, considera-se que o código desenvolvido é modular e facilmente extensível. A título de exemplo, devido à estrutura intermédia emitida pelo *parser*, mais rica em informação do que uma AST vulgar, foi trivial a implementação de geração de código para a EWVM, bem como também seria a adição de outros *backends*, como, por exemplo, para *output* de código C [7] ou LLVM [8]. Ademais, o código Python escrito foi todo tipado, e estes tipos verificados com o `mypy` [6]. A informação de tipos constitui uma grande fonte de documentação que, juntamente com nomes de variáveis descritivos, permite uma compreensão fácil do significado de diversos métodos, de campos de estruturas de dados, *etc.*

Por último, o compilador desenvolvido é bastante eficiente, na medida que foi arquitetado para minimizar o número de iterações pelas várias estruturas com que lida (código fonte, grafo intermédio, e representação intermédia do *asssembly* EWVM). O compilador também é capaz de gerar código eficiente, sendo usadas técnicas de manipulação do GST e de otimização *peephole* para melhorar o desempenho do código. Note-se que as otimizações feitas são bastante limitadas, visto que otimizações mais complexas exigiriam a construção de grafos de controlo de fluxo, e o desenvolvimento de outras representações intermédias, possivelmente da forma SSA, tal como o LLVM [8].

10 Conclusão

Em suma, ao longo do último semestre, foi desenvolvido, no âmbito da UC de Processamento de Linguagens, um compilador de Pascal Standard [1] para a *stack machine* EWVM [2]. Devido a falta de tempo, não foi possível explorar este problema tanto quanto era inicialmente desejado: no início deste projeto, considerou-se a possibilidade de se suportar o *standard* ISO completo, bem como mais *backends* além de EWVM. No entanto, aprendeu-se que a implementação de um compilador não é nada simples: apesar da especificação de Pascal ser curta e utilizar uma linguagem pouco complexa, ao contrário de outras linguagens, ter o cuidado de suportar os vários erros e *edge-cases* possíveis consumiu a maior parte do tempo de desenvolvimento. Ademais, foi necessário garantir que todas as funcionalidades conseguiam coexistir umas com as outras sem conflitos, garantindo uma geração de *assembly* correta independentemente do

código Pascal escrito. Por último, com o desenvolvimento deste compilador, percebeu-se o motivo pelo qual as técnicas de otimização são geralmente feitas em representações intermédias na forma SSA, tendo-se chegado à conclusão de que as otimizações *peephole* e por manipulação da árvore de sintaxe abstrata são bastante limitadas. Apesar do compilador desenvolvido não ter atingido os objetivos inicialmente planeados, cumpre os requisitos do enunciado do trabalho prático e implementa várias outras funcionalidades adicionais, pelo que se considera que este projeto foi concluído com sucesso.

11 Bibliografia

[1] *Information Technology – Programming Languages – Pascal*, ISO 7185, 1990. [Online]. Available: `https://archive.org/details/iso-iec-7185-1990-Pascal`

[2] S. Rocha, J. Ramalho. "Virtual Machine EWVM." EWVM. Accessed: May 29, 2025. [Online.] Available: `https://ewvm.epl.di.uminho.pt/`

[3] D. Beazley, "PLY (Python Lex-Yacc)." Dabeaz. Accessed: May 29, 2025. [Online.] Available: `https://www.dabeaz.com/ply/index.html`

[4] Free Software Foundation, "GCC online documentation". GCC, the GNU Compiler Collection. Accessed: May 31, 2025. [Online.] Available: `https://gcc.gnu.org/onlinedocs/gcc/Diagnostic-Message-Formatting-Options.html`

[5] P. Eaton, "Parser generators vs. handwritten parsers: surveying major language implementations in 2021." Phil Eaton. Accessed: May 31, 2025. [Online.] Available: `https://notes.eatonphil.com/parser-generators-vs-handwritten-parsers-survey-2021.html`

[6] The mypy project."mypy." mypy. Accessed: May 29, 2025. [Online.] Available: `https://mypy-lang.org/`

[7] *Information Technology – Programming Languages – C*, ISO 9899, 2024. [Online]. Available: `https://www.iso.org/standard/82075.html`

[8] LLVM-admin team."The LLVM Compiler Infrastructure." LLVM. Accessed: May 29, 2025. [Online.] Available: `https://llvm.org/`

12 Anexos

12.1 Lexemas

Segue-se a definição dos lexemas por ordem de prioridade de reconhecimento. Asteriscos azuis sinalizam que o lexema sofre alguma forma de transformação antes de ser passado ao analisador sintático.

```
PROGRAM      = r'PROGRAM\b'
BEGIN        = r'BEGIN\b'
END          = r'END\b'
LABEL        = r'LABEL\b'
CONST        = r'CONST\b'
TYPE         = r'TYPE\b'
VAR          = r'VAR\b'
ARRAY        = r'ARRAY\b'
PACKED       = r'PACKED\b'
SET          = r'SET\b'
FILE         = r'FILE\b'
OF           = r'OF\b'
RECORD       = r'RECORD\b'
FUNCTION     = r'FUNCTION\b'
PROCEDURE    = r'PROCEDURE\b'
IF           = r'IF\b'
THEN         = r'THEN\b'
ELSE         = r'ELSE\b'
FOR          = r'FOR\b'
TO           = r'TO\b'
DOWNTO       = r'DOWNTO\b'
DO           = r'DO\b'
WHILE        = r'WHILE\b'
REPEAT       = r'REPEAT\b'
UNTIL        = r'UNTIL\b'
CASE         = r'CASE\b'
GOTO         = r'GOTO\b'
WITH         = r'WITH\b'
AND          = r'AND\b'
OR           = r'OR\b'
NOT          = r'NOT\b'
IN           = r'IN\b'
DIV          = r'DIV\b'
MOD          = r'MOD\b'
NUL          = r'NIL\b'
ID           = r'[a-z][a-z0-9]*'
FLOAT        = r'[0-9]+(\.[0-9]+e(\+|\-)?[0-9]+|\.[0-9]+|e(\+|\-)?[0-9]+)\b' *
INTEGER      = r'[0-9]+\b' *
```

```
STRING       = r'\'((?:\'\'|[^\'])*)\'' *
ALT_CARET    = r'@' *
ALT_LBRACKET = r'\(\.' *
ALT_RBRACKET = r'\.\)' *
DIFFERENT    = r'<>'
LE           = r'<='
GE           = r'>='
ASSIGN       = r':='
RANGE        = r'\.\.'
'.'          = r'\.'
';'          = r';'
':'          = r':'
'('          = r'\('
','          = r','
')'          = r'\)'
'<'          = r'<'
'>'          = r'>'
'='          = r'='
'+'          = r'\+'
'-'          = r'-'
'*'          = r'\*'
'/'          = r'/'
'['          = r'\['
']'          = r'\]'
'^'          = r'\^'

ignore         =  ' \t\r'
ignore_COMMENT = r'({|\(\*)((?!{|\(\*)(.|\n))*?(}|\*\))'
```

12.2 Gramática

Segue-se a gramática utilizada no compilador, sem as produções utilizadas para deteção de erros:

```
program : PROGRAM ID program-arguments ';' block '.'

program-arguments :
                  | '(' identifier-list ')'

identifier-list : ID
                | identifier-list ',' ID

block : any-block-list begin-end-statement

any-block-list :
```

```
              | any-block-list-non-empty

any-block-list-non-empty : any-block
                         | any-block-list any-block

any-block : label-block
          | constant-block
          | type-block
          | variable-block
          | callable-block

label-block : LABEL label-list ';'

label-list  : INTEGER
            | label-list ',' INTEGER

constant-block : CONST constant-definition-list

constant-definition-list : constant-definition
                         | constant-definition-list constant-definition

constant-definition : ID '=' constant ';'

constant : unsigned-constant
         | signed-constant

unsigned-constant : ID
                  | unsigned-constant-literal

unsigned-constant-literal : FLOAT
                          | INTEGER
                          | STRING

signed-constant : '+' unsigned-constant
                | '-' unsigned-constant

type-block : TYPE type-definition-list

type-definition-list : type-definition
                     | type-definition-list type-definition

type-definition : ID '=' type ';'

type : type-id
     | pointer-type
     | enumerated-type
     | structured-type
     | range-type
```

```
type-id : ID

pointer-type : '^' type-id

enumerated-type : '(' identifier-list ')'

range-type : constant RANGE constant

structured-type : unpacked-structured-type
                | PACKED unpacked-structured-type

unpacked-structured-type : array-type
                         | record-type
                         | set-type
                         | file-type

array-type : ARRAY '[' array-dimensions ']' OF type

array-dimensions : range-type
                 | array-dimensions ',' range-type

record-type : RECORD ... END

set-type : SET OF type

file-type : FILE OF type

variable-block : VAR variable-definition-list

variable-definition-list : variable-definition
                         | variable-definition-list variable-definition

variable-definition : identifier-list ':' type ';'

variable-usage : ID variable-index-list

variable-index-list :
                    | variable-indices

variable-indices : variable-index
                 | variable-indices variable-index

variable-index : '[' expression-list ']'

callable-block : callable-list ';'

callable-list : callable-definition
              | callable-list callable-definition
```

```
callable-definition : procedure-heading ';' block
                    | function-heading ';' block

procedure-heading : PROCEDURE ID new-scope parameter-list

function-heading : FUNCTION ID new-scope parameter-list ':' type-id

new-scope :

parameter-list :
               | '(' parameter-list-non-empty ')'

parameter-list-non-empty : parameter
                         | parameter-list-non-empty ';' parameter

parameter : identifier-list ':' type-id

callable-call : ID callable-arguments

callable-arguments :
                   | '(' expression-list ')'

expression : non-relational-expression
           | non-relational-expression relational-operator non-relational-expression

non-relational-expression : first-term
                          | non-relational-expression adding-operator term

first-term : first-factor
           | first-term multiplying-operator factor

first-factor : '+' factor
             | '-' factor
             | factor

term : factor
     | term multiplying-operator factor

factor : '(' expression ')'
       | NIL
       | NOT factor
       | unsigned-constant-literal
       | ID variable-index-list
       | ID callable-arguments

relational-operator : '='
                    | DIFFERENT
                    | '<'
                    | '>'
```

```
                    | LE
                    | GE
                    | IN

adding-operator : '+'
                | '-'
                | OR

multiplying-operator : '*'
                     | '/'
                     | DIV
                     | MOD
                     | AND

expression-list : expression
                | expression-list ',' expression

begin-end-statement : BEGIN statement-list END

statement-list : statement
               | statement-list ';' statement

statement : unlabeled-statement
          | INTEGER ':' unlabeled-statement

unlabeled-statement :
                    | variable-usage ASSIGN expression
                    | variable-usage '=' expression
                    | GOTO INTEGER
                    | callable-call
                    | begin-end-statement
                    | IF expression THEN statement if-statement-else
                    | CASE expression OF case-element-list END
                    | CASE expression OF case-element-list ';' END
                    | REPEAT statement-list UNTIL expression
                    | WHILE expression DO statement
                    | FOR ID ASSIGN expression TO     expression DO statement
                    | FOR ID ASSIGN expression DOWNTO expression DO statement
                    | WITH variable-list DO statement

if-statement-else :
                  | ELSE statement

case-element-list : case-element
                  | case-element-list ';' case-element

case-element : constant-list ':' statement

constant-list : constant
```

```
              | constant-list ',' constant

variable-list : variable-usage
              | variable-list ',' variable-usage
```

12.3 Geração de Código dos *Statements*

`var := expr`	`expr` `var(mode = write)`
`GOTO label`	`JUMP label`

`procedure(expr1, expr2, ...)`
NOTA: `procedure` é definido pelo utilizador
	`expr1()`
	`expr2()`
	`...`
	`PUSHA procedure`
	`CALL`
	`POP N`

`IF expr THEN`
`  when-true;`
`ELSE`
`  when-false;`
	`expr()`
	`JZ else`
	`  when-true()`
	`  JUMP end`
	`else:`
	`  when-false()`
	`end:`

`REPEAT`
`  statements`
`UNTIL expr`
	`start:`
	`  statements()`
	`  expr()`
	`  JZ start`

`WHILE expr DO`
`  statement`
	`start:`
	`  expr()`
	`  JZ end`
	`  statement()`
	`  JUMP start`
	`end:`

```
FOR var := expr1 (TO|DOWNTO) expr2 DO
    statement
```

```
    expr2()
    expr1()
start:
    DUP 1
    var(mode = write)
    COPY 2
    (SUPEQ|INFEQ)
    JZ end
    statement()
    PUSHI 1
    (ADD|SUB)
    JUMP start
end:
    POP 2
```